USENIX

Proceedings of the 4th Annual
Linux Showcase & Conference, Atlanta

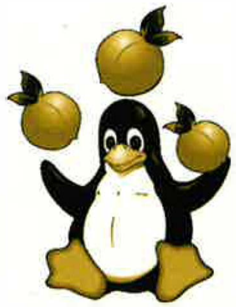
Atlanta, Georgia, USA, October 2000

conference

proceedings

4th Annual Linux Showcase & Conference, Atlanta

Atlanta, Georgia, USA
October 10–14, 2000



Sponsored by

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

and

The Atlanta Linux Showcase

in cooperation with

Linux International

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
URL: http://www.usenix.org

The price is $27 for members and $35 for nonmembers.
Outside the U.S.A. and Canada, please add
$14 per copy for postage (via air printed matter).

ISBN 1-880446-17-0
Printed in the United States of America on 50% recycled paper, 10–15% post-consumer waste.

USENIX Association

Proceedings of the 4th Annual Linux Showcase and Conference, Atlanta

October 10–14, 2000
Atlanta, Georgia, USA

Conference Organizers

Hack Linux: Technical Development Track

Chair: Theodore Ts'o, *VA Linux Systems*
Bryan C. Andregg, *Red Hat*
Peter Beckman, *TurboLinux, Inc.*
Jes Sorensen, *Linux Care*
Stephen Tweedie, *Red Hat*
Victor Yodaiken, *New Mexico Institute of Mining and Technology*
Leonard Zubkoff, *VA Linux Systems*

External Reviewer, Hack Linux
Jamal Hadi Salim, *Nortel Networks*

Extreme Linux: Clusters and High Performance Computing Workshop

Co-Chairs: Pete Beckman, *TurboLinux, Inc.,* and David Greenberg, *Center for Computing Sciences*
David Bader, *University of New Mexico*
Don Becker, *Scyld Computing Corporation and USRA-CESDIS*
Peter J. Braam, *Stelias Computing Inc. & Carnegie Mellon University*
Rajkumar Buyya, *Monash University, Australia*
Rémy Evard, *Argonne National Laboratories*
Jon "maddog" Hall, *VA Linux Systems*
David Halstead, *Ames Laboratory*
Yutaka Ishikawa, *RWCP*
Walter Ligon, *Clemson University*
Greg Lindahl, *HPTi*
Bill Nitzberg, *NASA*
Bill Saphir, *LBL/NERSC*
Matt Welsh, *UC–Berkeley*

Use Linux: Tools and Applications Track

Chair: Greg Hankins, *Atlanta Linux Showcase*
Eric Ayers, *Atlanta Linux Enthusiasts*
Jeff Carr, *LinuxPPC*
Clem Cole, *Compaq*
Danny Cox, *Atlanta Linux Showcase*
Hunter Eidson, *Atlanta Linux Showcase*
Jon "maddog" Hall, *VA Linux Systems*
Vernard Martin, *Atlanta Linux Showcase*

The USENIX Association Staff

4th Annual Linux Showcase & Conference, Atlanta

October 10–14, 2000
Atlanta, Georgia, USA

See http://www.linuxshowcase.org/
for the full program

Message from the Conference Chairs

In your hands you hold the results of the hard work of literally thousands, if not millions, of people. It contains the work of developers, researchers, and hobbyists who love Linux and take joy in exploring how to move Linux and the Open Source community to the next level.

It contains the work of the many volunteers on the Extreme Linux, Hack Linux, and Use Linux Program Committees who spent long hours reading and deliberating in order to select the very best papers and presentations from the over one hundred submissions we received.

Of course, we must also not overlook the hard work of Ellie Young, Jane-Ellen Long, Dana Geffner, Monica Ortiz, the many other USENIX staff members, and the volunteers who make up the Atlanta Linux Showcase, as well as those members of the Atlanta Linux Enthusiasts who worked tirelessly to make this 4th Annual Linux Showcase and Conference a logistical reality.

And finally, and most importantly, it contains the love and the hard work of all of you who are reading this set of proceedings today. For the Linux and Open Source communities would not be where they are today without you. The strength of Linux and Open Source fundamentally derives from every single member in the community.

We have very much enjoyed working on this small contribution to the Linux community. We hope you get at least as much pleasure out of reviewing all the technical, social, and business insights contained within these pages as we have had assembling it for you.

Peter Beckman

David Greenberg

Greg Hankins

Theodore Ts'o

Program Chairs

Index of Authors

Analyzing the Overload Behavior of a Simple Web Server

Niels Provos, *University of Michigan*
provos@citi.umich.edu

Chuck Lever, *AOL-Netscape, Incorporated*
chuckl@netscape.com

Stephen Tweedie, *Red Hat Software*
sct@redhat.com

Linux Scalability Project
Center for Information Technology Integration
University of Michigan, Ann Arbor

linux-scalability@citi.umich.edu
http://www.citi.umich.edu/projects/linux-scalability

Abstract

Linux introduces POSIX Real Time signals to report I/O activity on multiple connections with more scalability than traditional models. In this paper we explore ways of improving the scalability and performance of POSIX RT signals even more by measuring system call latency and by creating bulk system calls that can deliver multiple signals at once.

1. Introduction

Experts on network server architecture have argued that servers making use of I/O completion events are more scalable than today's servers [2, 3, 5]. In Linux, POSIX Real-Time (RT) signals can deliver I/O completion events. Unlike traditional UNIX signals, RT signals carry a data payload, such as a specific file descriptor that just completed. Signals with a payload can enable network server applications to respond immediately to network requests, as if they were event-driven. An added benefit of RT signals is that they can be queued in the kernel and delivered to an application one at a time, in order, leaving an application free to pick up I/O completion events when convenient.

The RT signal queue is a limited resource. When it is exhausted, the kernel signals an application to switch to polling, which delivers multiple completion events at once. Even when no signal queue overflow happens, however, RT signals may have inherent limitations due to the number of system calls needed to manage events on a single connection. This number may not be critical if the queue remains short, for instance while server workload is easily handled. When the server becomes loaded, the signal queue can cause system call overhead to dominate server processing, with the result that events are forced to wait a long time in the signal queue.

This document was written as part of the Linux Scalability Project. The work described in this paper was supported via generous grants from the Sun-Netscape Alliance, Intel, Dell, and IBM.

Linux has been carefully designed so that system calls are not much more expensive than library calls. There are no more cache effects for a system call than there are for a library call, and few virtual memory effects because the kernel appears in every process's address space. However, added security checks during system calls and hardware overhead caused by crossing protection domains make it expedient to avoid multiple system calls when fewer will do.

Process switching is still comparatively expensive, often resulting in TLB flushes and virtual memory overhead. If a system call must sleep, it increases the likelihood that the kernel will switch to a different process. By lowering the number of system calls required to accomplish a given task, we reduce the likelihood of harm to an application's cache resident set.

Improving the scalability and reducing the overhead of often-used system calls has a direct impact on the scalability of network servers [1, 4]. Reducing wait time for blocking system calls gives multithreaded server applications more control over when and where requested work is done. Combining several functions into fewer system calls has the same effect.

In this paper, we continue work begun in "Scalable Network I/O for Linux" by Provos, *et al.* [9]. We measure the effects of system call latency on the performance and scalability of a simple web server based on an RT signal event model. Of special interest is the way server applications gather pending RT signals. Today applications use `sigwaitinfo()` to dequeue pending signals one at a time. We create a new interface, called `sigtimedwait4()`, that is capable of delivering multiple signals to an application at once.

We use `phhttpd` as our web server. `Phhttpd` is a static-content caching front end for full-service web servers such as Apache [8]. Brown created `phhttpd` to demonstrate the POSIX RT signal mechanism, added to Linux during the 2.1 development series and completed during the 2.3 series [2]. We drive our test server with `httperf` [6]. An added client creates high-latency, low-bandwidth connections, as in Banga and Druschel [7].

Section 2 introduces POSIX Real-Time signals and describes how server designers can employ them. It also documents the `phhttpd` web server. Section 3 motivates the creation of our new system call. We describe our benchmark in Section 4, and discuss the results of the benchmark in Section 5. We conclude in Section 6.

2. POSIX Real-Time Signals and the `phhttpd` Web Server

In this section, we introduce POSIX Real-Time signals (RT signals), and provide an example of their use in a network server.

2.1 Using SIGIO with non-blocking sockets

To understand how RT signals provide an event notification mechanism, we must first understand how signals drive I/O in a server application. We recapitulate Stevens' illustration of signal-driven I/O here [10].

An application follows these steps to enable signal-driven I/O:

1. The application assigns a `SIGIO` signal handler with `signal()` or `sigaction()`.
2. The application creates a new socket via `socket()` or `accept()`.
3. The application assigns an owner pid, usually its own pid, to the new socket with `fcntl(fd, F_SETOWN, newpid)`. The owner then receives signals for this file descriptor.
4. The application enables non-blocking I/O on the socket with `fcntl(fd, F_SETFL, O_ASYNC)`.
5. The application responds to signals either with its signal handler, or by masking these signals and picking them up synchronously with `sigwaitinfo()`.

The kernel raises `SIGIO` for a variety of reasons:

- A connection request has completed on a listening socket.
- A disconnect request has been initiated.
- A disconnect request has completed.
- Half of a connection has been shut down.
- Data has arrived on a socket.
- A write operation has completed.
- An out-of-band error has occurred.

When using old-style signal handlers, this mechanism has no way to inform an application which of these conditions occurred. POSIX defines the `siginfo_t` struct (see FIG. 1), which, when used with the `sigwaitinfo()` system call, supplies a signal reason code that distinguishes among the conditions listed above. Detailed signal information is also available for new-style signal handlers, as defined by the latest POSIX specification [15].

This mechanism cannot say what file descriptor caused the signal, thus it is not useful for servers that manage more than one TCP socket at a time. Since its inception, it has been used successfully only with UDP-based servers [10].

2.2 POSIX Real-Time signals

POSIX Real-Time signals provide a more complete event notification system by allowing an application to associate signals with specific file descriptors. For example, an application can assign signal numbers larger than `SIGRTMIN` to specific open file descriptors using `fcntl(fd, F_SETSIG, signum)`. The kernel raises the assigned signal whenever there is new data to be read, a write operation completes, the remote end of the connection closes, and so on, as with the basic `SIGIO` model described in the previous section.

Unlike normal signals, however, RT signals can queue in the kernel. If a normal signal occurs more than once before the kernel can deliver it to an application, the kernel delivers only one instance of that signal. Other instances of the same signal are dropped. However, RT signals are placed in a FIFO queue, creating a stream of event notifications that can drive an application's response to incoming requests. Typically, to avoid complexity and race conditions, and to take advantage of the information available in `siginfo_t` structures, applications mask the chosen RT signals during normal operation. An application uses `sigwaitinfo()` or `sigtimedwait()` to pick up pending signals synchronously from the RT signal queue.

The kernel must generate a separate indication if it cannot queue an RT signal, for example, if the RT signal queue overflows, or kernel resources are temporarily exhausted. The kernel raises the normal signal `SIGIO` if this occurs. If a server uses RT signals to monitor incoming network activity, it must clear the RT signal queue and use another mechanism such as `poll()` to discover remaining pending activity when `SIGIO` is raised.

Finally, RT signals can deliver a payload. `Sigwaitinfo()` returns a `siginfo_t` struct (see FIG. 1) for each signal. The `_fd` and `_band` fields in this structure contain the same information as the `fd` and `revents` fields in a `pollfd` struct (see FIG. 2).

```
struct siginfo {
        int si_signo;
        int si_errno;
        int si_code;
        union {
                /* other members elided */
                struct {
                        int _band;
                        int _fd;
                } _sigpoll;
        } _sifields;
} siginfo_t;
```

Figure 1. Simplified `siginfo_t` struct.

```
struct pollfd {
        int fd;
        short events;
        short revents;
};
```

Figure 2. Standard `pollfd` struct

2.2.1 Mixing threads and RT signals

According to the GNU `info` documentation that accompanies `glibc`, threads and signals can be mixed reliably by blocking all signals in all threads, and picking them up using one of the system calls from the `sigwait()` family [16].

POSIX semantics for signal delivery do not guarantee that threads waiting in `sigwait()` will receive particular signals. According to the standard, an external signal is addressed to the whole process (the collection of all threads), which then delivers it to one particular thread. The thread that actually receives the signal is any thread that does not currently block the signal. Thus, only one thread in a process should wait for normal signals while all others should block them.

In Linux, however, each thread is a kernel process with its own PID, so external signals are always directed to one particular thread. If, for instance, another thread is blocked in `sigwait()` on that signal, it will not be restarted.

This is an important element of the design of servers using an RT signals-based event core. All normal signals should be blocked and handled by one thread. On Linux, other threads may handle RT signals on file descriptors, because file descriptors are "owned" by a specific thread. The kernel will always direct signals for that file descriptor to its owner.

2.2.2 Handling a socket close operation

Signals queued before an application closes a connection will remain on the RT signal queue, and must be processed and/or ignored by applications. For instance, when a socket closes, a server application may receive previously queued read or write events before it picks up the close event, causing it to attempt inappropriate operations on the closed socket.

When a socket is closed on the remote end, the local kernel queues a `POLL_HUP` event to indicate the remote hang-up. `POLL_IN` signals occurring earlier in the event stream usually cause an application to read a socket, and when it does in this case, it receives an EOF. Applications that close sockets when they receive `POLL_HUP` must ignore any later signals for that socket. Likewise, applications must be prepared for reads to fail at any time, and not depend only on RT signals to manage socket state.

Because RT signals queue unlike normal signals, server applications cannot treat these signals as interrupts. The kernel can immediately re-use a freshly closed file descriptor, confusing an application that then processes (rather than ignores) `POLL_IN` signals queued by previous operations on an old socket with the same file descriptor number. This introduces to the unwary application designer significant vulnerabilities to race conditions.

2.3 Using RT Signals in a Web Server

`Phhttpd` is a static-content caching front end for full-service web servers such as Apache [2, 8]. Brown created `phhttpd` to demonstrate the POSIX RT signal mechanism, added to the Linux kernel during the 2.1.x kernel development series and completed during the 2.3.x series. We describe it here to document its features and design, and to help motivate the design of `sigtimedwait4()`. Our discussion focuses on how `phhttpd` makes use of RT signals.

2.3.1 Assigning RT signal numbers

Even though a unique signal number could be assigned to each file descriptor, `phhttpd` uses one RT signal number for all file descriptors in all threads for two reasons.

1. Lowest numbered RT signals are delivered first. If all signals use the same number, the kernel always delivers RT signals in the order in which they arrive.
2. There is no standard library interface for multithreaded applications to allocate signal numbers atomically. Allocating a single number once during startup and giving the same number to all threads alleviates this problem.

2.3.2 Threading model

`Phhttpd` operates with one or more worker threads that handle RT signals. Additionally, an extra thread is created for managing logs. A separate thread pre-populates the file data cache, if requested.

Instead of handling incoming requests with signals, `phhttpd` may use polling threads instead. Usually, though, `phhttpd` creates a set of RT signal worker threads, and a matching set of polling threads known as *sibling* threads. The purpose of sibling threads is described later.

Each RT signal worker thread masks off the file descriptor signal, then iterates, picking up each RT signal via `sigwaitinfo()` and processing it, one at a time. To reduce system call rate, `phhttpd` `read()`s on a new connection as soon as it has `accept()`ed it. Often, on high-bandwidth connections, data is ready to be read as soon as a connection is `accept()`ed. `Phhttpd` reads this data and sends a response immediately to prevent another trip through the "event" loop, reducing the negative cache effects of handling other work in between the accept and the read operations.
Because the read operation is non-blocking, it fails with `EAGAIN` if data is not immediately present. The thread proceeds normally back to the "event" loop in this case to wait for data to become available on the socket.

2.3.3 Load balancing

When more than one thread is available, a simple load balancing scheme passes listening sockets among the threads by reassigning the listener's owner via `fcntl(fd, F_SETOWN, newpid)`. After a thread accepts an incoming connection, it passes its listener to the next worker thread in the chain of worker threads. This mechanism requires that each thread have a unique pid, a property of the Linux threading model.

2.3.4 Caching responses

Because `phhttpd` is not a full-service web server, it must identify requests as those it can handle itself, or those it must pass off to its backing server. Local files that `phhttpd` can access are cached by mapping them and storing the map information in a hash, along with a pre-formed http response. When a cached file is requested, `phhttpd` sends the cached response via `write()` along with the mapped file data.

Logic exists to handle the request via `sendfile()` instead. In the long run, this may be more efficient for several reasons. First, there is a limited amount of address space per process. This limits the total number of cached bytes, especially because these bytes share the address space with the pre-formed responses, hash information, heap and stack space, and program text. Using `sendfile()` allows data to be cached in extended memory (memory addressed higher than one or two gigabytes). Next, as the number of mapped objects grows, mapping a new object takes longer. On Linux, finding an unused area of an address space requires at least one search that is linear in the number of mapped objects in that address space. Finally, creating these maps requires expensive page table and TLB flush operations that can hurt system-wide performance, especially on SMP hardware.

2.3.5 Queue overflow recovery

The original `phhttpd` web server recovered from signal queue overflow by passing all file descriptors owned by a signal handling worker thread to a pre-existing poll-based worker thread, known as its *sibling*. The sibling then cleans up the signal queue, polls over all the file descriptors, processes remaining work, and passes all the file descriptors back to the original signal worker thread.

On a server handling perhaps thousands of connections, this creates considerable overhead during a period when the server is already overloaded. We modified the queue overflow handler to reduce this overhead. The server now handles signal queue overflow in the same thread as the RT signal handler; sibling threads are no longer needed. This modification appears in `phhttpd` version 0.1.2. It is still necessary, however, to build a fresh `poll_fd` array completely during overflow processing. This overhead slows the server during overflow processing, but can be reduced by maintaining the `poll_fd` array concurrently with signal processing.

RT signal queue overflow is probably not as rare as some would like. Some kernel designs have a single maximum queue size for the entire system. If any aberrant application stops picking up its RT signals (the thread that picks up RT signals may cause a segmentation fault, for example, while the rest of the application continues to run), the system-wide signal queue will fill. All other applications on the system that use RT signals will eventually be unable to proceed without recovering from a queue overflow, even though they are not the cause of it.

It is well known that Linux is not a real-time operating system, and that unbounded latencies sometimes occur. Application design may also prohibit a latency upper bound guarantee. These latencies can delay RT signals, causing the queue to grow long enough that recovery is required even when servers are fast enough to handle heavy loads under normal circumstances.

3. New interface: `sigtimedwait4()`

To reduce system call overhead and remove a potential source of unnecessary system calls, we'd like the kernel to deliver more than one signal per system call. One mechanism to do this is implemented in the `poll()` system call. The application provides a buffer for a vector of results. The system call returns the number of results it stored in the buffer, or an error.

Our new system call interface combines the multiple result delivery of `poll()` with the efficiency of POSIX RT signals. The interface prototype appears in FIG. 3.

```
int sigtimedwait4(const sigset_t *set,
        siginfo_t *infos, int nsiginfos,
        const struct timespec *timeout);
```

Figure 3. `sigtimedwait4()` prototype

Like its cousin `sigtimedwait()`, `sigtimedwait4()` provides the kernel with a set of signals in which it is interested, and a timeout value that is used when no signals are immediately ready for delivery. The kernel selects queued pending signals from the signal set specified by `set`, and returns them in the array of `siginfo_t` structures specified by `infos` and `nsiginfos`.

Providing a buffer with enough room for only one `siginfo_t` struct forces `sigtimedwait4()` to behave almost like `sigtimedwait()`. The only difference is that specifying a negative timeout value causes `sigtimedwait4()` to behave like `sigwaitinfo()`. The same negative timeout instead causes an error return from `sigtimedwait()`.

Retrieving more than one single signal at a time has important benefits. First and most obviously, it reduces the average number of transitions between user space and kernel space required to process a single server request. Second, it reduces the number of times per signal the per-task signal spinlock is acquired and released. This improves concurrency and reduces cache ping-ponging on SMP hardware. Third, it amortizes the cost of verifying the user's result buffer, although some believe this is insignificant. Finally, it allows a single pass through the signal queue for all pending signals that can be returned, instead of a pass for each pending signal.

The `sigtimedwait4()` system call enables efficient server implementations by allowing the server to "compress" signals- if it sees multiple read signals on a socket, for instance, it can empty that socket's read buffer just once.

The `sys_rt_sigtimedwait()` function is a moderate CPU consumer in our benchmarks, according to the results of kernel EIP histograms. About three fifths of the time spent in the function occurs in the second critical section in FIG. 4.

```
spin_lock_irq(&current->sigmask_lock);
sig = dequeue_signal(&these, &info);
if (!sig) {
  sigset_t oldblocked = current->blocked;
  sigandsets(&current->blocked,
             &current->blocked, &these);
  recalc_sigpending(current);
  spin_unlock_irq(&current->
                       sigmask_lock);

  timeout = MAX_SCHEDULE_TIMEOUT;
  if (uts)
    timeout = (timespec_to_jiffies(&ts)
           + (ts.tv_sec || ts.tv_nsec));

  current->state = TASK_INTERRUPTIBLE;
  timeout = schedule_timeout(timeout);

  spin_lock_irq(&current->sigmask_lock);
  sig = dequeue_signal(&these, &info);
  current->blocked = oldblocked;
  recalc_sigpending(current);
}
spin_unlock_irq(&current->sigmask_lock);
```

Figure 4. This excerpt of the `sys_rt_sigtimedwait()` kernel function shows two critical sections. The most CPU time is consumed in the second critical section.

The `dequeue_signal()` function contains some complexity that we can amortize across the total number of dequeued signals. This function walks through the list of queued signals looking for the signal described in `info`. If we have a list of signals to dequeue, we can walk the signal queue once picking up all the signals we want.

4. Benchmark description

In this section, we measure and report several aspects of server performance.

Our test harness consists of two machines running Linux connected via a 100 Mb/s Ethernet switch. The workload is driven by an Intel SC450NX with four 500MHz Xeon Pentium III processors (512Kb of L2 cache each), 512Mb of RAM, and a pair of SYMBIOS 53C896 SCSI controllers managing several LVD 10KRPM drives. Our web server runs on custom-built hardware equipped with a single 400MHz AMD K6-2 processor, 64Mb of RAM, and a single 8G 7.2KRPM IDE drive. The server hardware is small so that we can easily drive the server into overload. We also want to eliminate any SMP effects on our server, so it has only a single CPU.

Our benchmark configuration contains only a single client host and a single server host, which makes the simulated workload less realistic. However, our benchmark results are strictly for comparing relative performance among our implementations. We believe the results also give an indication of real-world server performance.

A web server's static performance naturally depends on the size distribution of requested documents. Larger documents cause sockets and their corresponding file descriptors to remain active over a longer time period. As a result the web server and kernel have to examine a larger set of descriptors, making the amortized cost of polling on a single file descriptor larger. In our tests, we request a 1 Kbyte document, a typical `index.html` file from the `monkey.org` web site.

4.1 Offered load

Scalability is especially critical to modern network service when serving many high-latency connections. Most clients are connected to the Internet via high-latency connections, such as modems, whereas servers are usually connected to the Internet via a few high bandwidth, low-latency connections. This creates resource contention on servers because connections to high-latency clients are relatively long-lived, tying up server resources. They also induce a bursty and unpredictable interrupt load on the server [7].

Most web server benchmarks don't simulate high-latency connections, which appear to cause difficult-to-handle load on real-world servers [5]. We've added an extra client that runs in conjunction with the `httperf` benchmark to simulate these slower connections to examine the effects of our improvements on more realistic server workloads [6]. This client program opens a connection, but does not complete an http request. To keep the number of high-latency clients constant, these clients reopen their connection if the server times them out.

In previous work, we noticed server performance change as the number of inactive connections varied [9]. As a result of this work, one of the authors modified `phhttpd` to correct this problem. The latest version of `phhttpd` (0.1.2 as of this writing) does not show significant performance degradation as the number of inactive connections increases. Therefore, the benchmark results we present here show performance with no extra inactive connections.

There are several system limitations that influence our benchmark procedures. There are only a limited number of file descriptors available for single processes; `httperf` assumes that the maximum is 1024. We modified `httperf` to cope dynamically with a large number of file descriptors. Additionally, because we use only a single client and server in our test harness, we can have only about 60,000 open sockets at a single point in time. When a socket closes it enters the TIMEWAIT state for sixty seconds, so we must avoid reaching the port number limitation. We therefore run each benchmark for 35,000 connections, and then wait for all sockets to leave the TIMEWAIT state before we continue with the next benchmark run. In the following tests, we run `httperf` with 4096 file descriptors, and `phhttpd` with five thousand file descriptors.

4.2 Execution Profiling

To assess our modifications to the kernel, we use the EIP sampler built in to the Linux kernel. This sampler checks the value of the instruction pointer (EIP register) at fixed intervals, and populates a hash table with the number of samples it finds at particular addresses. Each bucket in the hash table reports the results of a four-byte range of instruction addresses.

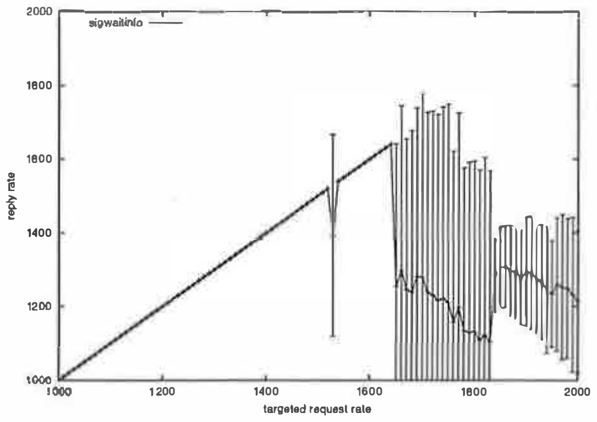
A user-level program later extracts the hash data and creates a histogram of CPU time matched against the kernel's symbol table. The resulting histogram demonstrates which routines are most heavily used, and how efficiently they are implemented. The granularity of the histogram allows us to see not only which functions are heavily used, but also where the most time is spent in each function.

5. Results and Discussion

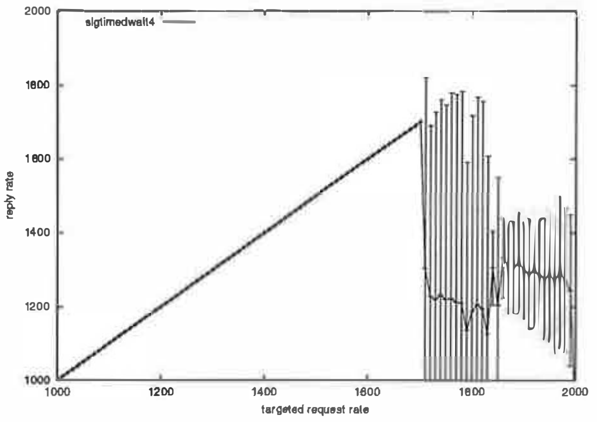
In this section we present the results of our benchmarks, and describe some new features that our new system call API enables.

5.1 Basic performance and scalability results

As described previously, our web server is a single processor host running a Linux 2.2.16 kernel modified to include our implementation of `sigtimedwait4()`. The web server application is phhttpd version 0.1.2. We compare an unmodified version with a version modified to use `sigtimedwait4()`. Our benchmark driver is a modified version of `httperf 0.6` running on a four-processor host.



Graph 1. Scalability of the `phhttpd` web server. This graph shows how a single threaded `phhttpd` web server scales as request rate increases. The axes are in units of requests per second.



Graph 2. Scalability of `phhttpd` using `sigtimedwait4()`. The signal buffer size was five hundred signals, meaning that the web server could pick up as many as five hundred events at a time. Compared to Graph 1, there is little improvement.

Our first test compares the scalability of unmodified `phhttpd` using `sigwaitinfo()` to collect one signal at a time with the scalability of `phhttpd` using `sigtimedwait4()` to collect many signals at once. The modified version of `phhttpd` picks up as many as 500 signals at once during this test.

Graphs 1 and 2 show that picking up more than one RT signal at a time gains little. Only minor changes in behavior occur when varying the maximum number of signals that can be picked up at once. The maximum throughput attained during the test increases slightly.

This result suggests that the largest system call bottleneck is not where we first assumed. Picking up signals appears to be an insignificant part of server overhead. We hypothesize that *responding to requests*, rather than picking them up, is where the server spends most of its effort.

5.2 Improving overload performance

While the graphs for `sigtimedwait4()` and `sigwaitinfo()` look disappointingly similar, `sigtimedwait4()` provides new information that we can leverage to improve server scalability.

Mogul, *et al.*, refer to "receive livelock," a condition where a server is not deadlocked, but makes no forward progress on any of its scheduled tasks [12]. This is a condition that is typical of overloaded interrupt-driven servers: the server appears to be running flat out, but is not responding to client requests. In general, receive livelock occurs because processing a request to completion takes longer than the time between requests.

Mogul's study finds that dropping requests as early as possible results in more request completions on overloaded servers. While the study recommends dropping requests in the hardware interrupt level or network protocol stack, we instead implement this scheme at the application level. When the web server becomes overloaded, it resets incoming connections instead of processing the requests.

To determine that a server is overloaded, we use a weighted load average, essentially the same as the TCP round trip time estimator [11, 13, 14]. Our new `sigtimedwait4()` system call returns as many signals as can fit in the provided buffer. The number of signals returned each time `phhttpd` invokes `sigtimedwait4()` is averaged over time. When the load average exceeds a predetermined value, the server begins rejecting requests.

Instead of dropping requests at the application level, using the listen backlog might allow the kernel to drop connections even before the application becomes involved in handling a request. Once the backlog overflows, the server's kernel can refuse connections, not even passing connection requests to the server application, further reducing the workload the web server experiences. However, this solution does not handle bursty request traffic gracefully. A moving average such as the RTT estimator smoothes out temporary traffic excesses, providing a better indicator of server workload over time.

The smoothing function is computed after each invocation of `sigtimedwait4()`. The number of signals picked up by `sigtimedwait4()` is one of the function's parameters:

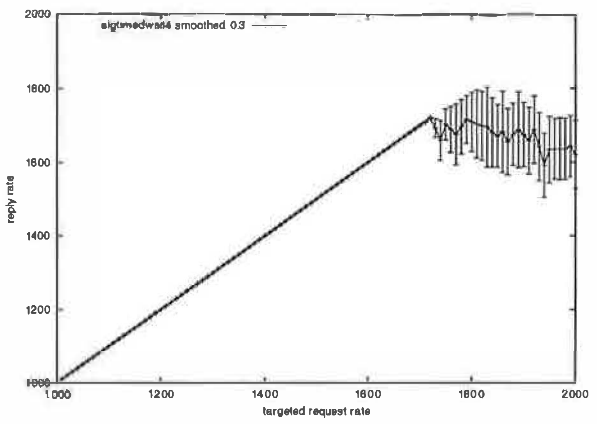
$$Avg_t = \alpha S + (1-\alpha)Avg_{t-1}$$

where S is the number of signals picked up by the most recent invocation of `sigtimedwait4()`; Avg is the moving load average; α is the gain value, controlling how much the current signal count influences the load average; and t is time.

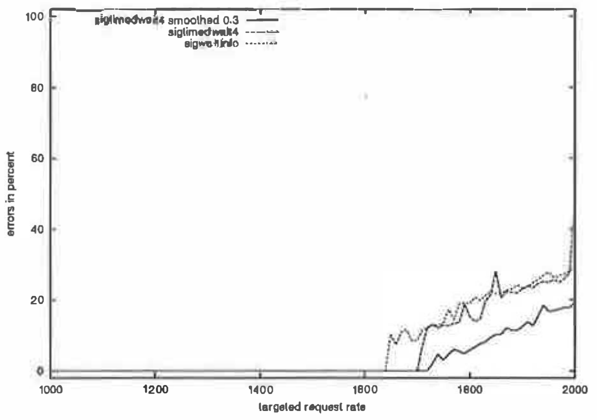
In our implementation, `phhttpd` picks up a maximum of 23 signals. If Avg exceeds 18, `phhttpd` begins resetting incoming connections. Experimentation and the following reasoning influenced the selection of these values. As the server picks up fewer signals at once, the sample rate is higher but the sample quantum is smaller. Only picking up one signal, for example, means we're either overloaded, or we're not. This doesn't give a good indication of the server's load. As we increase the signal buffer size, the sample rate goes down (it takes longer before the server calls `sigtimedwait4()` again), but the sample quantum improves. At some point, the sample rate becomes too slow to adequately detect and handle overload. That is, if we pick up five hundred signals at once, the server either handles or rejects connections for all five hundred signals.

The gain value determines how quickly the server reacts to full signal buffers (our "overload" condition). When the gain value approaches 1, the server begins resetting connections almost immediately during bursts of requests. Reducing the gain value allows the server to ride out smaller request bursts. If it is too small, the server may fail to detect overload, resulting in early performance degradation. We found that a gain value of 0.3 was the best compromise between smooth response to traffic bursts and overload reaction time.

Graphs 3 and 4 reveal an improvement in overload behavior when an overloaded server resets connections immediately instead of trying to fulfill the requests. Server performance levels off then declines slowly, rather than dropping sharply. In addition, connection error rate is considerably lower.



Graph 3. Scalability of `phhttpd` with averaged load limiting. Overload behavior improves considerably over the earlier runs, which suggests that formulating and sending responses present much greater overhead for the server than handling incoming signals.



Graph 4. Error rate of `phhttpd` with averaged load limiting. When the server drops connections on purpose, it actually reduces its error rate.

6. Conclusions and Future Work

Using `sigtimedwait4()` enables a new way to throttle web server behavior during overload. By choosing to reset connections rather than respond to incoming requests, our modified web server survives considerable overload scenarios without encountering receive livelock. The `sigtimedwait4()` system call also enables additional efficiency: by gathering signals in bulk, a server application can "compress" signals. For instance, if the server sees multiple read signals on a socket, it can empty that socket's read buffer just once.

Further, we demonstrate that more work is done during request processing than in handling and dispatching incoming signals. Lowering signal processing overhead in the Linux kernel has little effect on server performance, but reducing request processing overhead in the web server produces a significant change in server behavior.

It remains to be seen whether this request processing latency is due to:

- accepting incoming connections (`accept()` and `read()` system calls)
- writing the response (nonblocking `write()` system call and accompanying data copy operations)
- managing the cache (server-level hash table lookup and `mmap()` system call)
- some unforeseen problem.

Even though sending the response back to clients requires a copy operation, it is otherwise nonblocking. Finding the response in the server's cache should also be fast, especially considering the cache in our test contains only a single document. Thus we believe future work in this area should focus on the performance of the system calls and server logic that accept and perform the initial read on incoming connections.

This paper considers server performance with a single thread on a single processor to simplify our test environment. We should also study how RT signals behave on SMP architectures. Key factors influencing SMP performance and scalability include thread scheduling policies, the cache-friendliness of the kernel implementation of RT signals, and how well the web server balances load among its worker threads.

6.1. Acknowledgements

The authors thank Peter Honeyman and Andy Adamson for their guidance. We also thank the reviewers for their comments. Special thanks go to Zach Brown for his insights, and to Intel Corporation for equipment loans.

7. References

[1] G. Banga and J. C. Mogul, "Scalable Kernel Performance for Internet Servers Under Realistic Load," *Proceedings of the USENIX Annual Technical Conference*, June 1998.

[2] Z. Brown, *phhttpd*, `www.zabbo.net/phhttpd`, November 1999.

[3] Signal driven IO (*thread*), linux-kernel mailing list, November 1999.

[4] G. Banga. P. Druschel. J. C. Mogul. "Better Operating System Features for Faster Network Servers," *SIGMETRICS Workshop on Internet Server Performance*, June 1998.

[5] J. C. Hu, I. Pyarali, D. C. Schmidt, "Measuring the Impact of Event Dispatching and Concurrency Models on Web Server Performance Over High-Speed Networks," *Proceedings of the 2nd IEEE Global Internet Conference*, November 1997.

[6] D. Mosberger and T. Jin, "httperf – A Tool for Measuring Web Server Performance," *SIGMETRICS Workshop on Internet Server Performance, June 1998.*

[7] G. Banga and P. Druschel, "Measuring the Capacity of a Web Server," *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.

[8] Apache Server, The Apache Software Foundation. `www.apache.org`.

[9] N. Provos and C. Lever, "Scalable Network I/O in Linux," *Proceedings of the USENIX Technical Conference, FREENIX track,* June 2000.

[10] W. Richard Stevens, *UNIX Network Programming, Volume I: Networking APIs: Sockets and XTI*, 2nd edition, Prentice Hall, 1998.

[11] W. Richard Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*, pp. 299-309, Addison Wesley professional computing series, 1994.

[12] J. C. Mogul, K. K. Ramakrishnan, "Eliminating Receive Livelock in an Interrupt-driven Kernel," *Proceedings of USENIX Technical Conference*, January 1996.

[13] P. Karn and C. Partridge, "Improving Round-Trip Time Estimates in Reliable Transport Protocols," *Computer Communication Review*, pp. 2-7, vol. 17, no. 5, August 1987.

[14] V. Jacobson, "Congestion Avoidance and Control," *Computer Communication Review*, pp. 314-329, vol. 18, no. 4, August 1988.

[15] 1003.1b-1993 POSIX – Part 1: API C Language – Real-Time Extensions (ANSI/IEEE), 1993. ISBN 1-55937-375-X.

[16] GNU `info` documentation for `glibc`.

Appendix A: Man page for **sigtimedwait4()**

```
SIGTIMEDWAIT4(2)        Linux Programmer's Manual       SIGTIMEDWAIT4(2)

NAME
     sigtimedwait4 - wait for queued signals

SYNOPSIS
     #include <signal.h>

     int sigtimedwait4(const sigset_t *set, siginfo_t *infos,
           int nsiginfos, const struct timespec *timeout);

     typedef struct siginfo {
        int            si_signo; /* signal from signal.h */
        int            si_code;  /* code from above      */
      ...
        int            si_value;
      ...
     } siginfo_t;

     struct timespec {
        time_t         tv_sec;   /* seconds */
        long           tv_nsec;  /* and nanoseconds */
     };
DESCRIPTION
     sigtimedwait4() selects queued pending signals from the  set
     specified by set,  and   returns   them  in  the  array  of
     siginfo_t structs   specified   by   infos   and  nsiginfos.
     When multiple signals are pending,  the lowest numbered ones
     are selected.  The selection order between realtime and non-
     realtime signals, or between multiple  pending  non-realtime
     signals, is unspecified.

     sigtimedwait4() suspends  itself  for  the   time   interval
     specified in the timespec  structure  referenced by timeout.
     If  timeout  is  zero-valued,   or  no  timespec  struct  is
     specified, and if none of  the  signals  specified by set is
     pending, then sigtimedwait4()  returns  immediately with the
     error EAGAIN.   If  timeout   contains  a negative value, an
     infinite timeout is specified.

     If no signal in set is pending  at  the time  of  the  call,
     sigtimedwait4() suspends  the  calling  process until one or
     more signals  in  set  become  pending,  until  it is inter-
     rupted by an unblocked, caught signal,  or until the timeout
     specified   by   the   timespec   structure  pointed  to  by
     timeout expires.

     If, while sigtimedwait4() is waiting, a signal  occurs which
     is eligible for delivery (i.e., not blocked by  the  process
     signal mask), that  signal  is  handled  asynchronously  and
     the wait is interrupted.

     If  infos  is  non-NULL,  sigtimedwait4()  returns  as  many
     queued signals  as  are  ready  and  will  fit  in the array
     specified by infos. In  each  siginfo_t struct, the selected
     signal number is stored in si_signo,  and the cause  of  the
     signal is stored in the si_code. If a payload is queued with
     the signal, the payload value is stored in si_value.

     If the value  of  si_code  is  SI_NOINFO,  only the si_signo
     member of a siginfo_t struct is meaningful, and the value of
     all other members of that siginfo_t struct is unspecified.
     If no further signals are queued for  the  selected  signal,
     the pending indication for that signal is reset.

RETURN VALUES
     sigtimedwait4() returns  the  count  of siginfo_t structs it
     was able to store  in  the  buffer  specified  by  infos and
     nsiginfos.  Otherwise,  the  function  returns  -1 and sets
     errno to indicate any error condition.

ERRORS
     EINTR     The wait was interrupted by an  unblocked,  caught
               signal.

     ENOSYS    sigtimedwait4()   is   not   supported   by   this
               implementation.

     EAGAIN    No signal specified  by  set was delivered  within
               the specified timeout period.

     EINVAL    timeout specified a  tv_nsec  value less than 0 or
               greater than 1,000,000,000.

     EFAULT    The  array  of  siginfo_t  structs  specified  by
               infos and  nsiginfos  was  not  contained  in  the
               calling program's address space.

CONFORMING TO
     Linux

AVAILABILITY
     The  sigtimedwait4() system call  was  introduced  in  Linux
     2.4.

SEE ALSO
     time(2), sigqueue(2), sigtimedwait(2), sigwaitinfo(2)

Linux 2.4.0        Last change: 23 August 2000                 1
```

Linux Kernel Hash Table Behavior: Analysis and Improvements

Chuck Lever, *Sun-Netscape Alliance*
`<chuckl@netscape.com>`

Linux Scalability Project
Center for Information Technology Integration
University of Michigan, Ann Arbor

`linux-scalability@citi.umich.edu`
`http://www.citi.umich.edu/projects/linux-scalability`

Abstract

The Linux kernel stores high-usage data objects such as pages, buffers, and inodes in data structures known as hash tables. In this report we analyze existing static hash tables to study the benefits of dynamically sized hash tables. We find significant performance boosts with careful analysis and tuning of these critical kernel data structures.

1. Introduction

Hash tables are a venerable and well-understood data structure often used for high-performance applications because of their excellent average lookup time. Linux, an open-source POSIX-compliant operating system, relies on hash tables to manage pages, buffers, inodes, and other kernel-level data objects.

As we show, Linux performance depends on the efficiency and scalability of these data structures. On a small machine with 32M of physical RAM, a page cache hash table with 2048 buckets is large enough to hold all possible cache pages in chains shorter than three. However, a hash table this small cannot hold all possible pages on a larger machine with, say, 512M of physical RAM while maintaining short chains to keep lookup times quick. Keeping hash chains short is even more important on modern CPUs because of the effects of CPU cache pollution on overall system performance. Lookups on longer chains can expel useful data from CPU caches. The best compromise between fast lookup times on large-memory hardware and less wasted space on small machines is dynamically sizing hash tables as part of system start-up. The kernel can adjust the size of these hash tables depending on the conditions of the hardware at system boot time.

This document was written as part of the Linux Scalability Project. The work described in this paper was supported via generous grants from the Sun-Netscape Alliance, Intel, Dell, and IBM.

Hash tables depend on good average case behavior to perform well. Average case behavior relies on the actual input data more often than we like to admit, especially when using simple shift-add hash functions. In addition, hash functions chosen for statically allocated hash tables may be inappropriate for tables that can vary in size. Statistical examination of specific hash functions used in combination with specific real world data can expose opportunities for performance improvement.

It is also important to understand why hash tables are employed in preference to a more sophisticated data structure, such as a tree. Insertion into a hash table is **O(1)** if hashed objects are maintained in last-in first-out (LIFO) order in each bucket. A tree insertion or deletion is **O(*log* n)**. Object deletion and hash table lookup operations are often **O(n/m)** (where **n** is the number of objects in the table and **m** is the number of buckets), which approaches **O(1)** when the hash function spreads hashed objects evenly through the hash table and there are more hash buckets than objects to be stored. Finally, if designers are careful about hash table architecture, they can keep the average lookup time for both successful *and* unsuccessful lookups low (*i.e.* less than **O(*log* n)**) by using a large hash table and a hash function that thoroughly randomizes the key.

We want to know if larger or dynamically sized hash tables improve system performance, and if they do, by how much. In this report we analyze several critical hash tables in the Linux kernel, and describe minor tuning changes that can improve Linux performance by a considerable margin. We also show that current hash functions in the Linux kernel are, in general, appropriate for use with dynamically sized hash tables. The remainder of this report is organized as follows. Section 2 outlines our methodology. In Section 3, we separately examine four critical kernel hash tables and show how modifications to each hash table affect overall system performance. Section 4 reports results of combining the findings of Section 3. We discuss hash function theory as it applies on modern CPU architectures in Section 5, and Section 6 concludes the report.

2. Methodology

Our goal is to improve system throughput. Therefore, the final measure of performance improvement is benchmark throughput. However, there are a number of other metrics we can use to determine the "goodness" of a hash function with a given set of real-life input keys. In this section, we describe our benchmark procedures and the additional metrics we use to determine hash function goodness.

We use the SPEC SDM benchmark suite to drive our tests [7]. SDM emulates a multi-tasking software development workload with a fixed script of commands typically used by software developers, such as `cc`, `ed`, `nroff`, and `spell`. We model offered load by varying the number of simulated users, *i.e.*, concurrent instances of the script that run during the benchmark. The throughput values generated by the benchmark are in units of "scripts per hour." Each value is calculated by measuring the elapsed time for a given benchmark run, then dividing by the number of concurrent scripts running during the benchmark run. The elapsed time is measured in hundredths of a second.

We benchmark two hardware bases:

1. A Dell PowerEdge 6300/450 with 512M of RAM and a single Seagate 18G LVD SCSI hard drive. This machine uses four 450Mhz Xeon Pentium II processors, each with 512K of L2 cache.
2. A two processor custom-built system with 128M of RAM and a pair of 2G Quantum Fireball hard drives. This machine uses 200Mhz Pentium Pro CPUs with a 256K external L2 cache for each CPU, supported by the Intel i440FX chipset.

At the time of these tests, these machines were loaded with the Red Hat 5.2 distribution using a 2.2.5 Linux kernel built with egcs 1.1.1, and using glibc 2.0 as installed with Red Hat.

The dual Pentium Pro workloads vary from sixteen to sixty-four concurrent scripts. The sixteen-script workload fits entirely in RAM and is CPU bound. The sixty-four-script workload does not fit into RAM, thus it is bound by swap and file I/O. The four-way system ran up to 128 scripts before exhausting the system file descriptor limit because plain 2.2.5 kernels used in this report do not contain large `fdset` support. All of the 128 script benchmark fit easily into its 512M of physical memory, so this workload is designed to show how well the hash tables scale on large-memory systems when unconstrained by I/O and paging bottlenecks.

On our dual Pentium Pro, both disks are used for benchmark data and swap partitions. The swap partitions are of equal priority and size. The benchmark data is stored on file systems mounted with the "noatime" and "nosync" options for best performance. Likewise, on the four-way, the benchmark file system is mounted with the "noatime" and "nosync" options, and only one swap partition is used.

Hash performance depends directly on the laws of probability, so we are most interested in the statistical behavior of the hash (*i.e.* its "goodness"). First, we generate hash table bucket size distribution histograms with special kernel instrumentation. This tells us:

- What portion of total table buckets are unused,
- Whether a high percentage of hashed objects are contained in small buckets,
- The worst-case (largest) bucket size, and
- Whether bucket sizes are normally distributed. A normal distribution indicates that the hash function spreads objects evenly among the hash buckets, allowing the hash table to approach its best average behavior.

Second, we measure the average number of objects searched per bucket during lookup operations. This is a somewhat more general measure than elapsed time or instruction count because it applies equally to any hardware architecture. We count the average number of successful lookups separately from the average number of unsuccessful lookups because an unsuccessful lookup requires on average twice as many key comparisons. These search averages are one of the best indications of average bucket size and a direct measure of hash performance. Lowering them means better average hash performance.

Finally, we are interested in how long it takes to compute the hash function. This value is estimated given a table of memory and CPU cycle times, estimating memory footprint and access rate, cache miss rate, and guessing at how well the instructions to compute the hash function will be scheduled by the CPU. We estimate these based on Hennessey and Patterson [4].

3. Four critical hash tables

In this section we investigate the response to our tuning efforts of four critical kernel hash tables. These tables included the buffer cache, page cache, dentry cache, and inode cache hash tables.

3.1 Page cache

The Linux page cache contains in-core file data while the data is in use by processes running on the system. It can also contain data that has no backing storage, such as data in anonymous maps. The page cache hash table in the plain 2.2.5 kernel comprises 2048 buckets, and uses the following hash function from include/linux/pagemap.h:

```
#define PAGE_HASH_BITS 11
#define
    PAGE_HASH_SIZE (1 << PAGE_HASH_BITS)
```

```
static inline unsigned long
    _page_hashfn(struct inode * inode,
        unsigned long offset)
{
#define i (((unsigned long) inode)/
        (sizeof(struct inode) &
          ~ (sizeof(struct inode) - 1)))
#define o (offset >> PAGE_SHIFT)
#define s(x) ((x)+((x)>>PAGE_HASH_BITS))

        return s(i+o) & (PAGE_HASH_SIZE-1);

#undef i
#undef o
#undef s
}
```

The hash function key is made up of two arguments: the `inode` and the `offset`. The `inode` argument is a memory address of the in-core inode that contains the data mapped into the requested page. The `offset` argument is a memory address of the requested page relative to a virtual address space. The result of the function is an index into the page cache hash table.

This simple shift-add hash function is surprisingly effective due to the pre-existing randomness of the `inode` address and `offset` arguments. Our tests reveal that bucket size remains acceptable as `PAGE_HASH_BITS` is varied from 11 to 16.

Normally, the `offset` argument is page-aligned, but when the page cache is doubling as the swap cache, the `offset` argument can contain important index-randomizing information in the lower bits. Stephen Tweedie suggests that adding `offset` again, unshifted, before computing `s()`, would improve bucket size distribution problems caused when hashing swap cache pages [1]. Our tests show that adding the unshifted value of `offset` reduces bucket size distribution anomalies at a slight but measurable across-the-board performance cost.

kernel	table size (buckets)	16 scripts	32 scripts	48 scripts	64 scripts	total elapsed
reference	2048	1864.7 s=3.77	1800.8 s=8.51	1739.9 s=3.61	1644.6 s=29.35	50 min 25 sec
13-bit	8192	1875.8 s=5.59	1834.0 s=3.71	1765.5 s=3.01	1683.3 s=17.39	49 min 43 sec
14-bit	16384	1877.2 s=5.35	1830.8 s=3.81	1770.5 s=3.84	1694.3 s=41.42	49 min 35 sec
15-bit	32768	1875.4 s=10.72	1832.4 s=3.97	1770.3 s=3.97	1691.2 s=20.05	49 min 36 sec
offset	16384	1880.0 s=2.78	1843.7 s=14.65	1774.5 s=4.30	1685.4 s=33.46	49 min 40 sec
mult	16384	1876.4 s=6.45	1836.8 s=6.45	1773.7 s=5.20	1691.7 s=25.32	49 min 29 sec
rbtree	N/A	1874.9 s=6.57	1817.0 s=5.59	1755.3 s=3.01	1670.8 s=17.26	50 min 3 sec

Table 1. Benchmark throughput comparison of different hash functions in the page cache hash table. This table compares the performance of several Linux kernels using differently tuned hash tables in the page cache. Total benchmark elapsed time shows the multiplicative hash function improves performance the most.

table size, in buckets	average throughput	average throughput, minus first run	maximum throughput	elapsed time
2048	4282.8 s=29.96	4295.2 s=11.10	4313.0	12 min 57 sec
8192	4387.3 s=23.10	4398.5 s=5.88	4407.5	12 min 40 sec
32768	4405.3 s=5.59	4407.4 s=4.14	4413.8	12 min 49 sec

Table 2. Benchmark throughput comparison of different hash table sizes in the page cache hash table. This table shows benchmark performance of our tweaked kernels on large memory hardware. This test shows how performance changes when the data structure is heavily populated, and the system is not swapping.

Table 1 shows relative throughput results for kernels built with hash table tuning modifications. The "reference" kernel is a plain 2.2.5 kernel with a 4000 entry process table. The "13-bit," "14-bit," and "15-bit" kernels are plain 2.2.5 kernel with a 4000 entry process table and a 13, 14, and 15-bit (8192, 16384, and 32768 buckets) page cache hash table. The "offset" kernel is just like the "14-bit" kernel, but whose page cache hash function looks like this:

```
    return s(i+o+offset) & (PAGE_HASH_SIZE-1);
```

The "mult" kernel is the "14-bit" kernel with a multiplicative hash function instead of the plain additive one:

```
    return ((((unsigned long)inode + offset) *
        2654435761UL) >> \
            (32 - PAGE_HASH_BITS)) &
                (PAGE_HASH_SIZE-1);
```

See the section on multiplicative hashing for more about how we derived this function.

Finally, the "rbtree" kernel was derived from a clean 2.2.5 kernel with a special patch applied, extracted from Andrea Arcangeli's 2.2.5-arca10 patch. This patch implements the page cache with per-inode red-black trees, a form of balanced binary tree, instead of a hash table [3].

We run each workload seven times, and take the results from the middle five runs. The results in Table 1 are averages and standard deviations for the middle five benchmark runs for each workload. The timing result is the total length of all the runs for that kernel, including the two runs out of seven that were ignored in the average calculations. Each set of runs for a given kernel is benchmarked on a freshly rebooted system. These are obtained on our dual Pentium Pro using sixteen, thirty-two, forty-eight, and sixty-four concurrent script workloads to show how performance changes between CPU bound and I/O bound workloads. We also want to push the system into swap to see how performance changes when the page cache is used as a swap cache.

According to our own kernel program counter profiling results, defining `PAGE_HASH_BITS` as 13 bits is enough to take `find_page()` out of the top kernel CPU users during most heavy VM loads on large-memory machines. However, increasing it further can help reduce the real elapsed time required for an average lookup, improving system performance even more. As one might expect, increasing the hash table size had little effect on smaller workloads. To show the effects of increased table size on a high-end machine, we ran 128 script benchmarks on our four-way 512M Dell PowerEdge. The kernels used in this test are otherwise unchanged reference kernels compiled with 4000 process slots. The results are averages of five runs on each kernel.

The gains in inter-run variance are significant for larger memory machines. It is also clear that overall performance improves for tables larger than 8192 buckets, although not to the same degree that it improves for a table size increase of 2048 to 8192 buckets.

The "rbtree" kernel performs better than the "reference" kernel. It also scores very well in inter-run variance. A big advantage of this implementation is that it is more space efficient, especially on small machines, as it doesn't require contiguous pages for a hash table. We predicted the "offset" kernel to perform better when the system was swapping, but it appears to perform worse than both the "mult" and the "14-bit" kernel on the heaviest workload. Finally, the "mult" kernel appears to have the smoothest overall results, and the shortest overall elapsed time.

Because of the overall goodness of the existing hash function, the biggest gain occurs when the page cache hash table size is increased. This has performance benefits for machines of all memory sizes; as hash table size increases, more pages are hashed into buckets that contain only a single page, decreasing average lookup time.

Increasing the page cache hash table's bucket count even further continues to improve performance, especially for large memory machines. However, for use on generic hardware, 13 bits accounts for 8 pages worth of hash table, which is probably the practical upper limit for small memory machines.

In the 2.2.16 kernel, the page cache hash table is dynamically sized during system start-up. A hash table size is selected based on the physical memory size of the hardware; table size is the total number of pages available in the system multiplied by the size of a pointer. For example, 64 megabytes of RAM translate to 16384 buckets on hardware that supports 4-byte pointers.

Of course, the number of pages that can be allocated contiguously for the table limits its size. Hence the hash function mask and bit shift value are computed based on the actual size of the hash table. The mask in our example is 16383, or 0x3fff, one less than the number of buckets in the table. The shift value is 15, the number of bits in 16384.

The computed size for this table is unnecessarily large. In general, this formula provides a bucket for every page on the system on smaller machines. Performance on a 64M system is likely limited by many other factors, including memory fragmentation resulting from contiguous kernel data structures. According to our measurement of 2.2.5 kernels, a hash table a half or even a quarter of that size can still perform well, and would save memory and lower address space fragmentation.

3.2 Buffer cache

Linux holds dirty data blocks about to be written to disk in its buffer cache. The buffer cache hash table in the plain 2.2.5 kernel comprises 32768 buckets, and uses this hash function from fs/buffer.c:

```
#define HASHDEV(dev)    ((unsigned int) (dev))

#define _hashfn(dev,block) \
        (((unsigned)(HASHDEV(dev)^block)) & \
                bh_hash_mask)
```

This function adds no randomness to either argument, simply xor-ing them together, and truncating the result.

Histogram 1 was obtained during several heavy runs of our benchmark suite on the dual Pentium Pro hardware configuration. Each histogram divides its output into several columns. First, the "buckets" column reports the observed number of buckets in the hash table containing "size" objects; there are 1037 buckets observed to contain a single buffer in this example. The "buffers" column reports how many buffers are found in buckets of that size, a product of the size and observed bucket count. The "sum-pct" column is the cumulative percentage of buffers contained in buckets of that size and smaller. In other words, in Histogram 1, 28% of all buffers in the hash table are stored in buckets containing 8 or fewer buffers, and 42% of all buffers were stored in buckets containing 15 or fewer buffers. The number of empty buckets in the hash table is the value reported in the "buckets" column for size 0.

The average bucket size for 37,000+ buffers stored in a 16384 bucket table should be about 3 (that is, **O(n/m)**, where **n** is the number of objects contained in the hash table, and **m** is the number of hash buckets). The largest bucket contains 116 buffers, almost 2 orders of magnitude more than the expected average, even though the hash table is less than twenty-six percent utilized (16384 total buckets minus 12047 empty buckets, divided by 16384 total buckets gives us 0.26471). At one point during the benchmark, the author observed buckets containing more than 340 buffers.

```
Apr 27 17:17:51 pillbox kernel: Buffer cache total lookups: 296481  (hit rate: 54%)
Apr 27 17:17:51 pillbox kernel:  hash table size is 16384 buckets
Apr 27 17:17:51 pillbox kernel:  hash table contains 37256 objects
Apr 27 17:17:51 pillbox kernel:  largest bucket contains 116 buffers
Apr 27 17:17:51 pillbox kernel:  find_buffer() iterations/lookup: 2155/1000
Apr 27 17:17:51 pillbox kernel:  hash table histogram:
Apr 27 17:17:51 pillbox kernel:    size  buckets  buffers  sum-pct
Apr 27 17:17:51 pillbox kernel:     0    12047        0       0
Apr 27 17:17:51 pillbox kernel:     1     1037     1037       2
Apr 27 17:17:51 pillbox kernel:     2      381      762       4
Apr 27 17:17:51 pillbox kernel:     3      295      885       7
Apr 27 17:17:51 pillbox kernel:     4      325     1300      10
Apr 27 17:17:51 pillbox kernel:     5      399     1995      16
Apr 27 17:17:51 pillbox kernel:     6      188     1128      19
Apr 27 17:17:51 pillbox kernel:     7      303     2121      24
Apr 27 17:17:51 pillbox kernel:     8      160     1280      28
Apr 27 17:17:51 pillbox kernel:     9      169     1521      32
Apr 27 17:17:51 pillbox kernel:    10      224     2240      38
Apr 27 17:17:51 pillbox kernel:    11       64      704      40
Apr 27 17:17:51 pillbox kernel:    12       49      588      41
Apr 27 17:17:51 pillbox kernel:    13       15      195      42
Apr 27 17:17:51 pillbox kernel:    14        3       42      42
Apr 27 17:17:51 pillbox kernel:    15        4       60      42
Apr 27 17:17:51 pillbox kernel:   >15      721    21398     100
```

Histogram 1. Full buffer cache using the old hash function. This histogram demonstrates how poorly the Linux buffer cache spreads buffers across the buffer cache hash table. Most of the buffers are stored in hash buckets that contain more than 15 other buffers. This slows benchmark throughput markedly.

kernel	table size	average throughput	avg throughput, minus first run	maximum throughput	elapsed time
reference	32768	4282.8 s=29.96	4295.2 s=11.10	4313.0	12 min 57 sec
mult, shift 16	32768	4369.3 s=19.35	4376.4 s=14.53	4393.2	12 min 45 sec
mult, shift 11	32768	4380.8 s=12.09	4382.8 s=11.21	4394.0	12 min 50 sec
shift-add	32768	4388.9 s=21.90	4397.2 s=11.70	4415.5	12 min 31 sec
mult, shift 11	16384	4350.5 s=99.75	4394.6 s=15.59	4417.2	12 min 41 sec
mult, shift 17	16384	4343.7 s=61.17	4369.9 s=17.39	4390.2	12 min 46 sec
shift-add	16384	4390.2 s=22.55	4399.6 s=8.52	4408.3	12 min 37 sec
mult, shift 18	8192	4328.9 s=16.61	4333.7 s=15.05	4349.6	12 min 41 sec
shift-add	8192	4362.5 s=13.37	4362.8 s=14.90	4382.3	12 min 45 sec

Table 3. Benchmark throughput comparison of different hash functions in the buffer cache hash table. We report the results of benchmarking several new buffer cache hash functions in this table. Using a sophisticated multiplicative hash function appears to boost overall system throughput the most.

After the benchmark is over, most of the buffers still reside in large buckets (see Histogram 2). Eighty-five percent of the buffers in this cache are contained in buckets with more than 15 buffers in them, even though there are 16167 empty buckets—an effective bucket utilization of less than two percent!

Clearly, a better hash function is needed for the buffer cache hash table. The following table compares benchmark throughput results from the reference kernel (unmodified 2.2.5 kernel with 4000 process slots, as above) to results obtained after replacing the buffer cache hash function with several different hash functions. Here is our multiplicative hash function:

```
#define _hashfn(dev,block) (((block) * \
    2654435761UL) >> SHIFT) & \
        bh_hash_mask)
```

We tested variations of this function (`SHIFT` value is fixed at 11, or varies depending on the table size). We also tried a shift-add hash function to see if the multiplicative hash was really best. The shift-add function comes from Peter Steiner, and uses a shift and subtract ((block << 7) - block) to effectively multiply by a Mersenne prime (block * 127) [1]. Multiplication by a Mersenne prime is easy to calculate, as it reduces to a subtraction and a shift operation.

```
#define _hashfn(dev,block) \
        (((block << 7) - block + (block >> 10) \
    + (block >> 18)) & \
        bh_hash_mask)
```

This series of tests consists of five runs of 128 concurrent scripts on the four-way Dell PowerEdge system. We report an average result for all five runs, and an average result without the first run. The five-run average and the total elapsed time show how good or bad the first run,

```
Apr 27 17:30:49 pillbox kernel: Buffer cache total lookups: 3548568 (hit rate: 78%)
Apr 27 17:30:49 pillbox kernel:  hash table size is 16384 buckets
Apr 27 17:30:49 pillbox kernel:  hash table contains 2644 objects
Apr 27 17:30:49 pillbox kernel:  largest bucket contains 80 buffers
Apr 27 17:30:49 pillbox kernel:  find_buffer() iterations/lookup: 1379/1000
Apr 27 17:30:49 pillbox kernel:  hash table histogram:
Apr 27 17:30:49 pillbox kernel:   size  buckets  buffers  sum-pct
Apr 27 17:30:49 pillbox kernel:     0    16167        0        0
Apr 27 17:30:49 pillbox kernel:     1      110      110        4
Apr 27 17:30:49 pillbox kernel:     2       10       20        4
Apr 27 17:30:49 pillbox kernel:     3        3        9        5
Apr 27 17:30:49 pillbox kernel:     4        1        4        5
Apr 27 17:30:49 pillbox kernel:     5        0        0        5
Apr 27 17:30:49 pillbox kernel:     6        3       18        6
Apr 27 17:30:49 pillbox kernel:     7        1        7        6
Apr 27 17:30:49 pillbox kernel:     8        6       48        8
Apr 27 17:30:49 pillbox kernel:     9        2       18        8
Apr 27 17:30:49 pillbox kernel:    10        1       10        9
Apr 27 17:30:49 pillbox kernel:    11        2       22       10
Apr 27 17:30:49 pillbox kernel:    12        3       36       11
Apr 27 17:30:49 pillbox kernel:    13        3       39       12
Apr 27 17:30:49 pillbox kernel:    14        3       42       14
Apr 27 17:30:49 pillbox kernel:    15        1       15       15
Apr 27 17:30:49 pillbox kernel:   >15       68     2246      100
```

Histogram 2. Buffer cache using the old hash function, after benchmark is complete. This histogram shows that, even after the benchmark completes, most buffers in the cache remain in hash buckets containing more than 15 other buffers. Additionally, 3,000+ buffers stored in about 220 buckets, although more than 16,000 empty buckets remain. Over time, buffers tend to congregate in large buckets, and system performance suffers.

which warms the system caches after a reboot, can be. The four-run average indicates steady-state operation of the buffer cache.

On a Pentium II with 512K of L2 cache, the shift-add hash shows a higher average throughput than the multiplicative variants. On CPUs with less pipelining, the race is somewhat closer, probably because the shift-add function, when performed serially, can sometimes take as long as multiplication. However, the shift-add function also has the lowest variance in this test, and the highest first-run throughput, making it a clear choice for use as the buffer cache hash function.

We also tested with smaller hash table sizes to demonstrate that buffer cache throughput can be maintained using fewer buckets. Our test results bear this out; in fact, often these functions appear to work better with fewer buckets. Reducing the size of the buffer cache hash table saves more than a dozen contiguous pages (in the existing kernel, this hash table already consumes a contiguous 32 pages).

Histogram 3 shows what a preferred bucket size distribution histogram looks like. These runs were made with the mult-11 hash function and a 16384-bucket hash table. This histogram snapshot was made at approximately the same points during the benchmark as the examples above. After the benchmark completes, the hash table returns to a nominal state. We can also see that the measured iterations per loop average is an order of magnitude less than with the original hash function.

We'd like to underscore some of the good statistical properties demonstrated in Histogram 3. First, the bucket size distributions shown in this histogram approach the shape of a normal distribution, suggesting that the hash function is doing a good job of randomizing the keys. The maximum height of the distribution occurs for buckets of size 3 (our expected average), which is about **n/m**, where **n** is the number of stored objects, and **m** is the number of buckets. A perfect distribution centers on the expected average, and has very short tails on either side, only one or two buckets. While the distribution in Histogram 3 is somewhat skewed, observations of tables that are even more full show that the curve becomes less skewed as it fills; that is, as the expected average grows away from zero, the shape of the size distribution more closely approximates the normal distribution. In all cases we've observed, the tail of the skew is fairly short, and there appear to be few degenerations of the hash (where one or more very large buckets appear).

Second, in both Histogram 3 and 4, about 68% of all buffers contained in the hash table are stored in buckets containing the expected average number of buffers or less. The expected standard deviation is sixty-eight percent of all samples. Lastly, the number of empty buckets in the first example above is only 12.4%, meaning more than 87% of all buckets in the table are used.

The 2.2.16 kernel sports a new buffer cache hash function. The new hash function is a fairly complex shift-add function that is intended to randomize the fairly regular values of device numbers and block values. It is difficult to arrive at a function that is statistically good for the buffer cache, because block number regularity varies with the geometry and size of disk drives.

```
Apr 27 18:14:50 pillbox kernel: Buffer cache total lookups: 287696  (hit rate: 54%)
Apr 27 18:14:50 pillbox kernel:  hash table size is 16384 buckets
Apr 27 18:14:50 pillbox kernel:  hash table contains 37261 objects
Apr 27 18:14:50 pillbox kernel:  largest bucket contains 11 buffers
Apr 27 18:14:50 pillbox kernel:  find_buffer() iterations/lookup: 242/1000
Apr 27 18:14:50 pillbox kernel:  hash table histogram:
Apr 27 18:14:50 pillbox kernel:   size  buckets  buffers  sum-pct
Apr 27 18:14:50 pillbox kernel:     0     2034        0        0
Apr 27 18:14:50 pillbox kernel:     1     3317     3317        8
Apr 27 18:14:50 pillbox kernel:     2     4034     8068       30
Apr 27 18:14:50 pillbox kernel:     3     3833    11499       61
Apr 27 18:14:50 pillbox kernel:     4     2082     8328       83
Apr 27 18:14:50 pillbox kernel:     5      712     3560       93
Apr 27 18:14:50 pillbox kernel:     6      222     1332       96
Apr 27 18:14:50 pillbox kernel:     7       78      546       98
Apr 27 18:14:50 pillbox kernel:     8       46      368       99
Apr 27 18:14:50 pillbox kernel:     9       19      171       99
Apr 27 18:14:50 pillbox kernel:    10        5       50       99
Apr 27 18:14:50 pillbox kernel:    11        2       22      100
Apr 27 18:14:50 pillbox kernel:    12        0        0      100
Apr 27 18:14:50 pillbox kernel:    13        0        0      100
Apr 27 18:14:50 pillbox kernel:    14        0        0      100
Apr 27 18:14:50 pillbox kernel:    15        0        0      100
Apr 27 18:14:50 pillbox kernel:   >15        0        0      100
```

Histogram 3. Full buffer cache using the mult-11 hash function. This histogram of buffer cache hash bucket sizes shows marked improvement. Most buffers reside in small buckets, thus most buffers in the buffer cache can be found after checking fewer than two or three other buffers in the same bucket.

```
Apr 27 18:27:19 pillbox kernel: Buffer cache total lookups: 3530977  (hit rate: 78%)
Apr 27 18:27:19 pillbox kernel:  hash table size is 16384 buckets
Apr 27 18:27:19 pillbox kernel:  hash table contains 2717 objects
Apr 27 18:27:19 pillbox kernel:  largest bucket contains 6 buffers
Apr 27 18:27:19 pillbox kernel:  find_buffer() iterations/lookup: 215/1000
Apr 27 18:27:19 pillbox kernel:  hash table histogram:
Apr 27 18:27:19 pillbox kernel:   size  buckets  buffers  sum-pct
Apr 27 18:27:19 pillbox kernel:     0    14302        0        0
Apr 27 18:27:19 pillbox kernel:     1     1555     1555       57
Apr 27 18:27:19 pillbox kernel:     2      442      884       89
Apr 27 18:27:19 pillbox kernel:     3       73      219       97
Apr 27 18:27:19 pillbox kernel:     4        5       20       98
Apr 27 18:27:19 pillbox kernel:     5        3       15       99
Apr 27 18:27:19 pillbox kernel:     6        4       24      100
Apr 27 18:27:19 pillbox kernel:     7        0        0      100
Apr 27 18:27:19 pillbox kernel:     8        0        0      100
Apr 27 18:27:19 pillbox kernel:     9        0        0      100
Apr 27 18:27:19 pillbox kernel:    10        0        0      100
Apr 27 18:27:19 pillbox kernel:    11        0        0      100
Apr 27 18:27:19 pillbox kernel:    12        0        0      100
Apr 27 18:27:19 pillbox kernel:    13        0        0      100
Apr 27 18:27:19 pillbox kernel:    14        0        0      100
Apr 27 18:27:19 pillbox kernel:    15        0        0      100
Apr 27 18:27:19 pillbox kernel:   >15        0        0      100
```

Histogram 4. Buffer cache using the mult-11 hash function, after the benchmark is complete. The reader can compare this histogram with the earlier one that reports the buffer cache bucket size distribution after the benchmark has completed. As buffers are removed from the buffer cache, the bucket size distribution remains good when using the multiplicative hash function.

The size of the buffer cache hash table is also computed dynamically during system start-up. Like the page cache hash table, the buffer cache hash table size is computed relative to the memory size of the host hardware. On a system with 64 megabytes of RAM, the computed hash table is 64K buckets. The hash function mask and bit shift values are computed like the same values for the page cache hash function.

Again, the computed size for this table is unnecessarily large. Each bucket requires two pointers because the buckets in this hash table are doubly-linked lists, so a 64K bucket table requires 256K of contiguous memory. The buffer cache hash table size is much too large for small memory configuration, and it doesn't grow much as memory size increases past 128M.

Our measurements show that, assuming the new hash function is reasonable, a much smaller table will still provide acceptable performance. A large table size is especially unnecessary in 2.4 and later kernels because write performance is not as dependent on the size of the buffer cache.

A comment near the table size computation logic notes that the table should be large enough to keep `fsync()` fast. This is a poor measure of table size, because it is well-known that `fsync()` is inefficiently implemented. A more reasonable way to help `fsync()` performance is to re-implement file syncing using a more efficient algorithm.

3.3 Dentry cache

The Linux 2.2 kernel has a directory entry cache, or *dentry* cache, that is designed to speed up file system performance by mapping file pathnames directly to the in-core address of the `inode` struct associated with the file. The plain 2.2.5 kernel uses a hash table with 1024 buckets to manage the dentry cache. A simple shift-add hash function is employed:

```
#define D_HASHBITS      10
#define D_HASHSIZE      (1UL << D_HASHBITS)
#define D_HASHMASK      (D_HASHSIZE-1)

static inline struct list_head * d_hash(
    struct dentry * parent,
        unsigned long hash)
{
    hash += (unsigned long) parent;
    hash = hash ^
        (hash >> D_HASHBITS) ^
        (hash >> D_HASHBITS*2);
    return dentry_hashtable +
        (hash & D_HASHMASK);
}
```

The arguments for this function are the address of the parent directory's dentry structure, and a hash value obtained by a simplified CRC algorithm on the target entry's name. This function appears to work fairly well, but we want to improve it nonetheless.

Andrea Arcangeli suggests that shrinking the dcache more aggressively might reduce the number of objects in the table enough to help improve dcache hash lookup times [1]. We test this idea by adding a couple of lines from his 2.2.5-arca10 patch: In fs/dcache.c, function `shrink_dcache_memory()`, we replace `prune_dcache(found)` with:

kernel	average throughput	elapsed time
reference	4282.8 s=29.96	12 min 57 sec
12 bit	4361.3 s= 11.15	12 min 36 sec
mult	4346.0 s=20.87	12 min 52 sec
14 bit	4368.3 s= 20.41	12 min 54 sec

Table 4. Benchmark throughput comparison of different hash functions in the dcache cache hash table. This table shows that increasing the hash table size in the dentry cache has significant benefits for system throughput, decreasing benchmark elapsed time by 15 seconds. Other changes decrease elapsed time by only a few seconds.

```
    prune_dcache(dentry_stat.nr_unused /
        (priority+1));
```

and in kswapd (the kernel's swapper daemon), we move the `shrink_dcache_memory()` call in `do_try_to_free_pages()` close to the top of the loop so that it will be invoked more often.

In Table 4, we show results from several different kernels. First, results from the reference 2.2.5 kernel are repeated from previous tables, then a kernel that is like the reference kernel, except the dcache hash table is increased to 16384 buckets, and the xor operations are replaced with addition when computing the hash function. The "shrink" kernel is a 2.2.5 kernel like the "14-bit" kernel except that it more aggressively shrinks the dcache, as explained above. The "mult" kernels use a multiplicative hash function similar to the buffer cache hash function, instead of the existing dcache hash function:

```
static inline struct list_head * d_hash(
    struct dentry * parent,
    unsigned long hash)
{
    hash += (unsigned long) parent;
    hash = (hash * 2654435761UL) >> SHIFT;
    return dentry_hashtable +
        (hash & D_HASHMASK);
}
```

where `SHIFT` is either 11 or 17. The "shrink+mult" kernels combine the effects of both multiplicative hashing and shrinking the dcache.

The results are averages from five benchmark runs of 128 concurrent scripts on the four-way Dell PowerEdge. The timing results are the elapsed time for all five runs on each kernel.

Some may argue that shrinking the dcache unnecessarily might lower the overall effectiveness of the cache, but we believe that shrinking the cache more aggressively will help, rather than hurt, overall system performance because a smaller cache allows faster lookups and causes less CPU cache pollution. In combination with an appropriate multiplicative hash function, such as the one used in the "shrink+mult 11" kernel, elapsed time and average throughput stays high enough to make it the fastest kernel benchmarked in this series.

The size of the dentry cache hash table in the 2.2.16 kernel is dynamically determined during system start-up. Like the previous two tables we examined, the hash table size is computed as a multiple of a system's physical memory size. On our imaginary 64-megabyte system, the dentry cache hash table contains 8192 buckets, and requires a 14-bit hash function shift value. This provides excellent performance without consuming excessive amounts of memory. There is also plenty of room to scale this table as memory size increases.

The dentry cache hash function in 2.2.16 computes an intermediate value modulus the hardware's L1 cache size. It is not clear whether this extra step improves the distribution of the hash function, since this filters noise that is already removed by the hash mask.

Dcache pruning appears no more aggressive in the 2.2.16 kernel than in earlier kernels. Some modifications to the swapper may improve the probability that `shrink_dcache_memory()` is invoked, however.

3.4 Inode cache

The dentry cache, described above, provides a fast way of mapping directory entries to inodes. Kernel developers expected the dentry cache to reduce the need for an efficient inode cache. Thus, when the dentry cache was implemented, the inode cache hash table was reduced to 256 buckets (8 bit hash). As we shall see, this has had a more profound impact on system performance than expected.

The inode cache hash function is a shift-add function similar to the dentry cache hash function.

```
#define HASH_BITS       8
#define HASH_SIZE       (1UL << HASH_BITS)
#define HASH_MASK       (HASH_SIZE-1)

static inline unsigned long hash(
    struct super_block *sb,
    unsigned long i_ino)
{
    unsigned long tmp = i_ino |
        (unsigned long) sb;
    tmp = tmp + (tmp >> HASH_BITS) +
        (tmp >> HASH_BITS*2);
    return tmp & HASH_MASK;
}
```

kernel	average throughput	maximum throughput	elapsed time
reference	4282.8 s=29.96	4313.0	12 min 57 sec
14 bit	4375.2 s=25.92	4397.4	12 min 42 sec
mult, shift 11	4368.7 s=62.65	4406.2	12 min 39 sec
mult, shift 17	4375.9 s=10.40	4389.0	12 min 40 sec
shrink	4368.7 s=33.36	4390.7	12 min 40 sec
shrink + mult 11	4380.4 s=13.53	4396.5	12 min 35 sec
shrink + mult 17	4368.5 s=16.21	4383.6	12 min 42 sec

Table 5. Benchmark throughput comparison of different hash functions in the inode cache hash table. Increasing the size of the inode cache hash table has clear performance benefits, as this table shows. Replacing the hash function in this cache actually hurts performance.

Histogram 5 shows why this table is too small. The hash chains are extremely long. In addition, the hit rate shows that most lookups are unsuccessful, meaning that almost every lookup request has to traverse the entire bucket. The average number of iterations per lookup is almost 40!

Even though there are an order of magnitude fewer lookups in the inode cache than there are in the other caches, this cache is still clearly a performance bottleneck. To demonstrate this, we ran tests on four different hash functions. Our reference kernel results (from Table 1) reappear in Table 5 for convenience. The "12-bit" kernel is the same as the reference kernel except that the hash table size has been increased to 4096 buckets. The "mult" kernel has 4096 inode cache hash table buckets as well, and uses the multiplicative hash function introduced above. The "14-bit" kernel is the same as the reference kernel except that the hash table size has been increased to 16384 buckets.

The 12-bit hash table is the clear winner. Increasing the hash table size further helps performance slightly, but also increases inter-run variance to such an extent that total elapsed time is longer than for the "12-bit" kernel. Adding multiplicative hashing doesn't help much here because the table is already full, and well balanced.

There is no difference between the 2.2.5 inode cache hash table implementation and the implementation that appears in the 2.2.16 kernel. Simply making this hash table larger by a factor of four would be an effective performance and scalability improvement for 2.2.16. The inode cache hash table size is dynamically computed in 2.4 kernels during system start-up. The 2.4 kernel's inode cache can grow considerably larger than earlier versions, thus it requires a scalable hash table.

```
Apr 27 17:23:31 pillbox kernel: Inode cache total lookups: 189321  (hit rate: 3%)
Apr 27 17:23:31 pillbox kernel:  hash table size is 256 buckets
Apr 27 17:23:31 pillbox kernel:  hash table contains 9785 objects
Apr 27 17:23:31 pillbox kernel:  largest bucket contains 54 inodes
Apr 27 17:23:31 pillbox kernel:  find_inode() iterations/lookup: 38978/1000
Apr 27 17:23:31 pillbox kernel:  hash table histogram:
Apr 27 17:23:31 pillbox kernel:   size  buckets    inodes sum-pct
Apr 27 17:23:31 pillbox kernel:     0        0         0       0
Apr 27 17:23:31 pillbox kernel:     1        0         0       0
Apr 27 17:23:31 pillbox kernel:     2        0         0       0
Apr 27 17:23:31 pillbox kernel:     3        0         0       0
Apr 27 17:23:31 pillbox kernel:     4        0         0       0
Apr 27 17:23:31 pillbox kernel:     5        0         0       0
Apr 27 17:23:31 pillbox kernel:     6        0         0       0
Apr 27 17:23:31 pillbox kernel:     7        0         0       0
Apr 27 17:23:31 pillbox kernel:     8        0         0       0
Apr 27 17:23:31 pillbox kernel:     9        0         0       0
Apr 27 17:23:31 pillbox kernel:    10        0         0       0
Apr 27 17:23:31 pillbox kernel:    11        0         0       0
Apr 27 17:23:31 pillbox kernel:    12        0         0       0
Apr 27 17:23:31 pillbox kernel:    13        0         0       0
Apr 27 17:23:31 pillbox kernel:    14        0         0       0
Apr 27 17:23:31 pillbox kernel:    15        0         0       0
Apr 27 17:23:31 pillbox kernel:   >15      256      9785     100
```

Histogram 5. Full inode cache using the old hash function. This histogram shows what happens when too many objects are stored in an undersized hash table. Every inode in this hash table resides in a bucket that contains, on average, 37 other objects. Combined with the very low hit rate, this results in a significant negative performance impact.

4. Combination testing

In this section, we optimize all hash tables we've studied so far, and benchmark the resulting kernels. Our benchmarks are ten 128 script runs on the four-way Dell.

We selected optimizations among the best results shown above, then tried them in combination. We find that there are performance relationships among the various caches, so we show the results for the best combinations that we tried.

The "Reference" kernel is a stock 2.2.5 Linux kernel with 4000 process slots:

- a 32768 bucket buffer hash table with a one-to-one hash function
- a 2048 bucket page hash table with a simple shift-add hash function
- a 256 bucket inode hash table with a simple shift-add hash function
- a 1024 bucket dentry hash table with a simple shift-add hash function

Kernel "A" is a plain 2.2.5 Linux kernel with 4000 process slots and:

- a 16384 bucket hash table using the multiply and shift-by-11 hash function
- a 8192 bucket page cache with the multiplicative hash function described in the page cache section
- a 2048 bucket inode hash table using a slightly modified shift-add hash function
- a 8192 bucket dcache hash table with addition instead of XOR in its hash function.

Kernel "B" is a plain 2.2.5 Linux kernel with 4000 process slots and:

- a 16384 bucket buffer hash table with Peter Steiner's shift-add hash function
- a 8192 bucket page cache with the multiplicative hash function described in the page cache section
- a 2048 bucket inode hash table using a slightly modified shift-add hash function
- a 8192 bucket dcache hash table with addition instead of XOR in its hash function.

Kernel "C" is a plain 2.2.5 Linux kernel with 4000 process slots and:

- a 16384 bucket hash table using the multiply and shift-by-11 hash function
- a 8192 bucket page cache with the reference kernel's hash function
- a 2048 bucket inode hash table using a slightly modified shift-add hash function
- a 8192 bucket dcache hash table with addition instead of XOR in its hash function.

Kernel "D" is a plain 2.2.5 Linux kernel with 4000 process slots and:

- a 16384 bucket has table using the multiply and shift-by-11 hash function
- a 8192 bucket page cache with the offset hash function described above
- a 2048 bucket inode hash table using a slightly modified shift-add hash function
- a 8192 bucket dcache hash table with addition instead of XOR in its hash function

Examining Table 6, we'd like to select a combination that reduces inter-run variance and elapsed time, as well as maximizes throughput and minimizes hash table memory footprint. While kernel "C" offers the highest maximum throughput, its inter-run variance is also largest. On the other hand, kernel "D" has the second highest average throughput, the shortest elapsed time, and the best inter-run variance. This seems like a reasonable compromise.

kernel	average throughput	maximum throughput	elapsed time
Reference	4300.7 s=15.73	4321.1	26 min 41 sec
Kernel A	4582.9 s=12.55	4592.8	25 min 24 sec
Kernel B	4577.9 s=16.22	4602.0	25 min 18 sec
Kernel C	4596.2 s=22.30	4619.5	25 min 18 sec
Kernel D	4591.3 s=10.98	4608.9	25 min 15 sec

Table 6. Benchmark throughput comparison of multiple kernel hash optimizations. Combining improvements in each of the four caches we studied results in an elapsed time improvement of almost a minute and a half.

5. Multiplicative hashing

Hash function alternatives include:

- Using an untransformed key
- Modulus hashing
- Multiplicative hashing
- Using an inexpensive but sub-optimal shift-add hash function
- Using a "correct" shift-add hash function
- Using a hash function driven by one or more random tables
- Architecture-specific hash functions (*e.g.* multiplication on fast, modern processors, and something else on older processors)

Multiplicative hashing is a form of modulus hashing that is less expensive because the results are often as good but a multiplication operation is used instead of a division operation. Multiplicative hashing is controversial because of the expense of multiplication instructions on some hardware types. For example, on 68030 CPUs, popular in old Sun and Macintosh computers, multiplication requires up to 44 CPU cycles for a 32-bit multiplication, whereas a memory load only requires an extra 2 cycles per instruction [8]. On a hardware architecture like the 68030 that has little caching, fast load times compared to CPU operations, and expensive multiplication, a multiplicative hash might be inferior even if it cuts the average number of loop iterations per lookup request by a factor of four or more.

However, it turns out that several of the alternatives are just as expensive, or even more expensive, than multiplicative hashing. Random table-driven hash functions require several table lookups, and several shifts, logical AND operations, and additions. An e-mail message from the linux-kernel mailing list explains the problem; see Appendix A.

On our example 68030, shifting requires between 4 and 10 cycles, and addition operations aren't free either. If the instructions that implement the hash function are many, they will likely cause instruction cache contention that will be worse for performance than a multiplication operation. In general, a proper shift-add hash function is almost as expensive in CPU cycles as a multiplicative hash. On a modern superscalar processor, shifting and addition operations can occur in parallel as long as there are no address generation interlocks (AGIs). An AGI occurs when the results of one operation are required to form an address in a later operation that might otherwise have been parallelized by superscalar CPU hardware [6, 9]. AGIs are much more likely for a table-driven hash function.

Multiplicative hash functions are often very concise. The hash functions we tried above, for example, compile to three instructions on ia32, comprising 15 bytes. Included in the 15 bytes are all the constants involved in the calculation, leaving only the key itself to be loaded as data. In other words, the whole hash function fits into a single line in the CPU's instruction cache on contemporary hardware. The shift-add hash functions are generally lengthy, requiring several cache lines to contain, multiple loads of the key, and register allocation contention.

The question becomes, finally, how many CPU cycles should be spent by the hash function to get a reasonable bucket size distribution? In most practical situations, a simple shift-add function suffices. However, one should always test with actual data before deciding on a hash function implementation. Hashing on block numbers, as the Linux buffer cache does, turns out to require a particularly good hash function, as disk block numbers exhibit a great deal of regularity.

5.1 A Little Theory

Our multiplicative hash functions were derived from Knuth, p. 513ff [5]. The theory posits that machine multiplication by a large number that is likely to cause overflow is the same as finding the modulus by a different number. We won't repeat Knuth here, but suffice it to say that choosing such a number is complicated. In brief, our choice is based on finding a prime that is in golden ratio to the machine's word size (2 to the 32nd in our case). Primality isn't strictly necessary, but it adds certain desirable qualities to the hash function. See Knuth for a discussion of these desirable qualities.

We selected 2,654,435,761 as our multiplier. It is prime, and its value divided by 2 to the 32nd is a very good approximation of the golden ratio [2, 10].

$$\frac{\sqrt{5}-1}{2} \cong 0.6180339887$$

$$\frac{2654435761}{2^{32}} \cong 0.6180339868$$

To obtain the best effects of this "division" we need to choose the correct shift value. This is usually the word size, in bits, minus the hash table size, in bits. This shifts the most significant bits of the result of the "division" down to where they can act as the hash table index, preserving the greatest effects of the golden ratio. Sometimes experimentation reveals a better shift value for a given set of input data, however.

6. Conclusions and Future Work

Careful selection and optimization of kernel hash tables can boost performance considerably, and improve inter-run variance as well, maximizing system throughput. Selecting a good hash function and benchmarking its effectiveness can be tedious, however. Usually, the most notable performance optimization comes from increasing the size of a hash table. In this report, we have shown that larger and/or dynamically sized hash tables are essential for Linux kernel performance and scalability. Adding dynamic hash table sizing is a simple way to get a five to 20% performance improvement, depending on how much physical memory is available on a system.

To extend this study, the cache instrumentation patch should be re-written to use a file in /proc instead of writing to system console log, and should be integrated into the stock kernel as a "Kernel Hacking" configuration option. The tuning patch should be benchmarked on 64-bit hardware to see if another constant must be chosen there. A benchmark run on older architectures, such as MC68000, should determine if these changes would seriously degrade performance on older machines.

We could also investigate the performance difference between in-lining the page cache management routines (which eliminates the subroutine call overhead) and leaving them as stand-alone routines (which means they have a smaller L1 cache footprint). A separate swap cache hash function might also optimize the separate uses of the page cache hash tables.

Additionally, there are still open questions about why shrinking the dentry cache more aggressively can help performance. A study could focus on the cost of a dentry cache miss versus the cost of a page fault or buffer cache miss. Discovering alternative ways of triggering a dentry cache prune operation, or alternate ways of calculating the prune priority, may also be interesting.

Finally, there is still opportunity to analyze even more carefully the real keys and hash functions in use in several of the tables we've analyzed here, as well as several tables we didn't visit in this report, such as the `uid` and `pid` hash tables, and the `vma` data structures.

For more information on modifications and kernel instrumentation described in this report, see the Linux Scalability Project web site:

`http://www.citi.umich.edu/projects/linux-scalability`

7. Acknowledgements

The author gratefully acknowledges the input and contributions of the following persons: Peter Steiner, Andrea Arcangeli, Iain McClatchie, Paul F. Dietz, Janos Farkas, Dr. Horst von Brand, and Stephen C. Tweedie, as well as the many others who contributed directly and indirectly to the work described in this report. Special thanks go to Dr. Charles Antonelli and Prof. Gary Tyson for providing the hardware benchmarked in this report. Thanks also to the reviewers for their input.

References

1. linux-kernel mailing list archives
2. *CRC Standard Mathematical Tables*, 25th Edition, William H. Beyer, Ed., CRC Press, Inc., 1978.
3. T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, 1990.
4. D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach,* 2nd Edition, Morgan Kaufmann, 1996.
5. D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd Ed., Addison-Wesley, 1998.
6. M. L. Schmit, *Pentium(tm) Processor Optimization Tools*, Academic Press, Inc., 1995.
7. Standard Performance Evaluation Corporation, System Development Multitask Benchmark SPEC, 1991
8. *MC68030 User's Manual, Volume 2*, Motorola, Incorporated, 1998.
9. Pentium II processor reference manuals, Intel Corporation.
10. *The Largest Known Primes*, `www.utm.edu/research/primes/largest.html`, 1998.

Appendix A: E-mail

```
Date: Thu, 15 Apr 1999 15:01:54 -0700
From: Iain McClatchie
To: Paul F. Dietz
Cc: linux-kernel@vger.rutgers.edu
Subject: Re: more on hash functions

I got a few suggestions about how to use multiple lookups with a
single table.  All the suggestions make the hash function itself
slower, and attempt to fix an issue -- hash distribution - that
doesn't appear to be a problem.  I thought I should explain why
the table lookup function is slow.

A multiplication has a scheduling latency of either 5 or 9 cycles on a
P6.  Four memory accesses take four cycles on that same P6.  So the core
operations for the two hash function are actually very similar in delay,
and the table lookup appears to have a slight edge.  The difference is
in the overhead.

A multiplicative hash, at minimum, requires the loading of a constant,
a multiplication, and a shift.  Egcs actually transforms some constant
multiplications into a sequence of shifts and adds which may have
shorter latency, but essentially, the shift (and nothing else) goes in
series with the multiplication and as a result the hash function has
very little latency overhead.

A table lookup hash spends quite a lot of time unpacking the bytes
from the key, and furthermore uses a load slot to unpack each byte.
This makes for 8 load slots, which take 1 cycle each.  Even if
fully parallelized with unpacking, we end up with a fair bit of
latency.  Worse yet, egcs runs out of registers and ends up shifting
the key value in place on the stack twice, which gobbles two load and
two store slots.

Bottom line: CPUs really suck at bit-shuffling and even byte-shuffling.
If there is some clever way to code the byte unpacking in the table
lookup hash function, perhaps using the x86's trick register file,
it might end up faster than the multiplicative hash.

-Iain
```

Dynamic Buffer Cache Management Scheme based on Simple and Aggressive Prefetching*

H. Seok Jeon   Sam H. Noh
Department of Computer Engineering
Hong-Ik University
Mapo-Gu Sangsoo Dong 72-1 Seoul, Korea 121-791
tel) +82-2-320-1470   fax)+82-2-320-1105
hsjeon@cs.hongik.ac.kr
samhnoh@cs.hongik.ac.kr, http://www.cs.hongik.ac.kr/~noh

Abstract

Many replacement and prefetching policies have recently been proposed for buffer cache management. However, many real operating systems, including GNU/Linux, generally use the simple Least Recently Used (LRU) replacement policy with prefetching being employed in special situations such as when sequentiality is detected. In this paper, we propose the SA-W^2R scheme that integrates buffer management and prefetching, where prefetching is done constantly in aggressive fashion. The scheme is simple to implement making it a feasible solution in real systems. In its basic form, for buffer replacement, it uses the LRU policy. However, its modular design allows for any replacement policy to be incorporated into the scheme. For prefetching, it uses the LRU-One Block Lookahead (LRU-OBL) approach, eliminating any extra burden that is generally necessary in other prefetching approaches. Implementation studies based on the GNU/Linux kernel version 2.2.14 show that the SA-W^2R performs better than the current version of GNU/Linux with a maximum increases of 23 % for the workloads considered.

1 Introduction

Considerable research for optimizing the use of buffer caches in both replacement and prefetching aspects have been undertaken. This paper proposes yet another scheme, but which has sailent features such as simple prefetching and modular integration of replacement and prefetching. These features lead to a scheme that is easily implementable, and which leads to performance improvements compared to previously known schemes.

In the following, we first discuss the state-of-the-art in buffer cache replacement and prefetching, and then point out their limitations, which is the motivation behind the development of the proposed scheme.

1.1 Previous Research in Buffer Cache Management

Many replacement policies for improving the performance of buffer cache management have been proposed. Policies such as the Least Recently Used (LRU), Least Frequently Used (LFU), LRU-K [11], 2Q [6], Frequency-Based Replacement (FBR) [14], and Least Recently/Frequently Used (LRFU) [8] are examples.

All these replacement policies were developed independent of prefetching. Recent developments in buffer management policies has lead to the investigation of incorporating prefetching to the replacement policies. It has been shown that for some workloads incorporating prefetching can result in considerable improvement in the performance of buffer management [15, 16].

Research in incorporating prefetching can be categorized into three groups. The first group of policies maintain a history of past behavior of the applications [5, 7, 9]. This speculative approach, however, may result in performance degradation due to inaccurate prefetching and history maintenance overhead.

The second category of policies obtain hints from applications themselves prior to execution or during execution [2, 3, 4, 10, 13, 17]. Currently, these approaches appear promising as it has been shown that hints may

*This work was supported by the Korea Science and Engineering Foundation grant 98-0102-09-01-3.

be obtainable for specific applications with minor overhead, though their feasibility in real systems still needs to be tested.

The final approach does not require any information neither from the application nor from observations of past behavior. The LRU-OBL (One Block Lookahead) scheme [15, 16] (and its variant of prefetching multiple blocks at once) is the only scheme known to date using this approach. This scheme simply prefetches the logical next block of the currently referenced block if it is not resident in cache. It has been shown that through this simple scheme, improvements of up to 80% in the hit rate was possible for some workloads [16]. To the best of our knowledge, this approach is the only prefetch technique used in real systems due to its simplicity and effectiveness.

1.2 Buffer Cache Management in GNU/Linux

GNU/Linux adopts the LRU scheme for buffer cache management. In GNU/Linux, the *bread()* function handles the block requests. If the requested block exists in the hash table, that is, the buffer cache, it returns the block pointer. Otherwise, it makes a request for a disk I/O.

The *breada()* function is provided as a primitive for prefetching in GNU/Linux. This function is simply a variant of the LRU-OBL scheme in that it reads blocks adjacent to the requested block. However, in the GNU/Linux kernel version 2.2.14, the *breada()* function is seldom used. To the best of our knowledge, prefetching is issued only at two places. One is for reading directories in the ISO 9660 file system and the other is for the kernel thread that synchronizes the spare disk with the active disk array in RAID reconstruction.

1.3 The Remainder of the Paper

The rest of the paper is organized as follows. In the next section, we provide the motivation behind this work. In Section 3, the SA-W^2R scheme, which is the buffer cache management scheme proposed in this paper, is presented. Simulation and implementation experimental studies are presented in Section 4 and 5, respectively. Finally, Section 6 concludes with a summary and directions for further research.

2 Motivation

In this section, we discuss the motivation behind the development of the SA-W^2R scheme that is presented in the next section. To this end, we first describe the Weighing-Waiting Room (W^2R) scheme. This scheme provides the framework for an efficient integration of buffer replacement and prefetching that is simple and effective. However, its limitation restricts it from being deployed in real systems hence, providing the basis for the development of the SA-W^2R scheme.

2.1 The W^2R Scheme

The Weighing-Waiting Room (W^2R) scheme partitions the buffer cache into two rooms, that is, the Weighing Room and the Waiting Room as shown in Figure 1. The name is derived from the fact that we use the weight analogy in describing the management of the buffer cache. In general, the block to be replaced by the incoming block is the block that is considered to be the least likely to be re-referenced. This likelihood can be represented as a weight. Each block is given a weight, and the heavier block is considered more likely to be re-referenced. Then, in general, the lightest block is replaced by the incoming block as it is considered to be the least likely to be referenced again.

The Weighing Room, in the W^2R scheme, is where the weights of the blocks are contested, and rank is formed among the blocks. Only blocks that have been referenced have weights associated with them. In buffer replacement policies such as the LRU or 2Q, the whole buffer is simply the Weighing Room as only blocks that have been referenced are brought into the buffer.

The Waiting Room is where the prefetched blocks reside. Prefetching is done exactly like the LRU-OBL scheme, where the logical next block of the currently accessed block is prefetched when it is not residing in cache. They remain in this part of the buffer and wait until they obtain permission to be weighed with the other blocks. This permission is obtained when and only when the block has actually been referenced. Until this time, the prefetched blocks are weightless. Again, blocks obtain weight only when referenced as their weight cannot be determined until they have been referenced.

Note some of the features of this scheme. First, replacement (through the Weighing Room) and prefetching (through the Waiting Room) are integrated into the whole scheme, yet modularity is ensured. New replace-

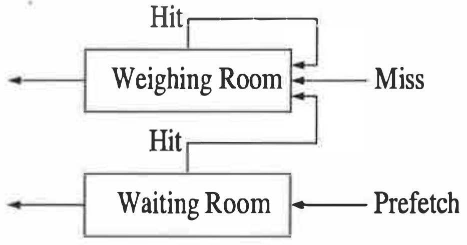


Figure 1: Structure of the W^2R scheme.

ment policies are constantly being developed. As new replacement policies that have practical significance are developed, these policies can directly be incorporated into the W^2R scheme, simply by replacing the implementation in the Weighing Room. This is unlike the LRU-OBL scheme that provide no means of doing this.

Second, by partitioning the buffer into a Weighing Room and a Waiting Room, prefetched blocks, which have not yet proven their worth, cannot replace a block in the Weighing Room, which has proven its value by being referenced. Figure 2 quantifies a deficiency in the LRU-OBL approach. This figure shows that in some situations over 60% of prefetched blocks may never be used when using the LRU-OBL scheme. Hence, in LRU-OBL, a prefetched block can hold on to valuable real estate without ever being referenced. This problem is alleviated in the W^2R scheme as blocks that are not referenced after being prefetched are not promoted to the Weighing Room. Hence, a prefetched block that is never referenced cannot replace a block that has some weight (that is, in the Weighing Room). That is to say, the wrong block may be prefetched into the buffer, but it will not replace a block that has proven its worth by having been referenced.

Third, note that a prefetched block enters the Weighing Room only after it is referenced. Hence, the prefetched block does not directly replace a block that is referenced prior to the prefetched block. The block being promoted to the Weighing Room is promoted right when it is needed, and never before.

2.2 Motivation for SA-W^2R

Although the benefits of the W^2R scheme seem to be attractive, as it stands, there is a serious problem that needs to be resolved in order for it to be deployable in real systems. The obvious and difficult problem is how to partition a fixed-size buffer cache into the two rooms. By introducing the Waiting Room, we are in effect, reducing the size of the Weighing Room compared to conventional buffer management policies. Hence, we

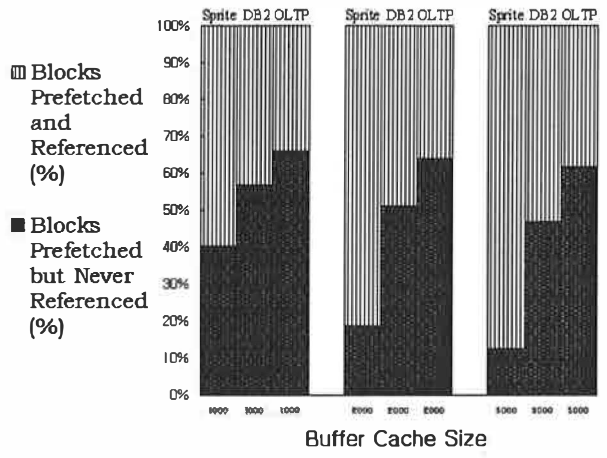


Figure 2: Percentage of prefetched blocks, using the LRU-OBL scheme, that are never referenced for the Sprite, DB2, and OLTP traces.

want to keep the Waiting Room small. However, once a block is prefetched we would like to hold it in the Waiting Room just enough so that it is eventually referenced. That is, if the Waiting Room is too small a prefetched block may be evicted too early to be of any help to the system. A judicious selection of the room size is necessary for efficient management of the buffer. This problem is addressed in the SA-W^2R scheme.

3 Self-Adjusting W^2R

The optimal partition ratio between the Weighing Room and the Waiting Room will vary according to the system environment and workload. To reiterate the point mentioned above, we want to keep the Waiting Room small so that the deterioration in performance due to the reduced Weighing Room size is limited, but we want to keep it big enough so that the prefetched blocks are used, instead of being evicted from the Waiting Room. Hence, the partitioning should accommodate the workload characteristics such that the performance benefits obtained by increasing the Waiting Room outweighs the loss incurred by the reduced Weighing Room. For all practical purposes, this should be determined on-line and must be done with minimal overhead. The Self-Adjusting W^2R (SA-W^2R) scheme attempts to do both.

The SA-W^2R adjusts the partitioning between the Weighing and Waiting Rooms via a two-step process, namely, interval-based and fault-based adjustment steps as shown in Figure 3. We discuss the two steps in

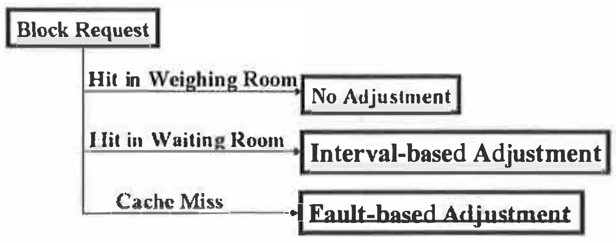


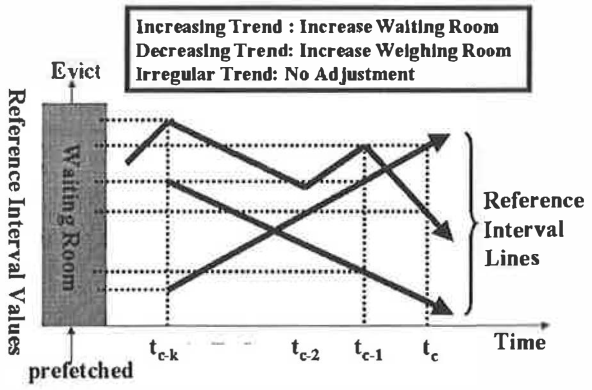
Figure 3: The SA-W^2R Scheme.

the following subsections.

3.1 Interval-based Adjustment

The Waiting Room, as the name implies, is where the prefetched blocks wait to be referenced. In other words, the role of the Waiting Room is to maintain the prefetched blocks until they are actually referenced. Recall that prefetched blocks are weightless, hence they are managed in a FIFO queue; the newly prefetched block is put at the rear (position 1) of the queue, which pushes the block at the head (position n) of the queue out of the FIFO queue, where n is the size of the Waiting Room. Let us define the position in the FIFO queue at which a block is actually referenced to be the *reference interval* of that block. If the block is evicted from the Waiting Room without being referenced, then the reference interval of that block is ∞. This reference interval will vary from workload to workload, and in adjusting the room sizes the adjustment should be such that the majority of the blocks have reference intervals less or equal to n for as small an n as possible. Hence, given a certain n value, if the reference intervals of blocks are getting smaller and smaller, then n should also be adjusted to become smaller, so that the loss in the Weighing Room can be minimized. On the other hand, if the reference intervals of blocks are getting larger and getting close to or surpasses n, then n should be increased, so as to increase the benefits of the Waiting Room.

The SA-W^2R scheme tunes the Waiting Room size by maintaining the reference interval values of the last k blocks that are referenced in the Waiting Room. (To minimize bookkeeping overhead, we set k to 3.) Using this information, we observe the trend of the reference intervals as shown in Figure 4. If the reference interval of the last k blocks show an increasing trend, then the Waiting Room is increased to accommodate the reference interval increase. Similarly, if the reference intervals show a decreasing trend, the Waiting Room is decreased. If the reference intervals of the last k blocks do not show any regularity, the Waiting Room size remains unchanged.



Increasing Trend : Increase Waiting Room
Decreasing Trend: Increase Weighing Room
Irregular Trend: No Adjustment
Evict
Reference Interval Values
Waiting Room
Reference Interval Lines
t_{c-k}
t_{c-2}
t_{c-1}
t_c
Time
prefetched

Figure 4: Interval-based Adjustment.

3.2 Fault-based Adjustment

Interval-based adjustments cannot be ideal as it adjusts the Waiting Room size based only on observations made in the Waiting Room. Adjustments of the two rooms can also be made based on hints provided by misses that occur upon a block reference. Consider the following situation. Block i is referenced, but it is not in the buffer, hence a miss occurs. Since we are prefetching the logical next block, we can deduce considerable information based on the availability of blocks $i-1$ and $i+1$ in the buffer cache. For example, if we find that block $i-1$ is in the Weighing Room and that block i is in disk, then we know that block i was evicted from the buffer cache as block i would have been prefetched with block $i-1$.

The SA-W^2R scheme exploits these fault-based information for adjusting the room sizes. It considers the situation when a request for block i is a miss. When a miss occurs on block i, nine possible situations can arise depending on the location of blocks i, $i-1$, and $i+1$ as shown in Table 1. Since a miss occurred on block i, it is in disk. At this point, blocks $i-1$ and $i+1$ can be in the Weighing Room, the Waiting Room, or the disk. Table 1 also shows the adjustments that are made for each of the nine cases.

Let us now consider how these adjustment recommendations came about. Let us start with cases 1 and 3, which have in common block $i-1$ in the Weighing Room. (Case 2 also falls into this category, but we will consider this case separately later.) The fact that block $i-1$ is in the Weighing Room tells us that block i had once been in the Waiting Room before being evicted. It may also have been in the Weighing Room before being evicted, but the fact that block $i+1$ is not in the Waiting Room makes it likely that block i was in the Waiting Room before being evicted. (Note that we are

Table 1: Nine situations in relation to the locations of blocks i, $i-1$, $i+1$ when a miss for block i occurs.

Cases	Weighing Room	Waiting Room	Disk	Adjustment Made
case 1	i-1, i+1		i	Increase Waiting Room
case 2	i-1	i+1	i	Increase Weighing Room
case 3	i-1		i, i+1	Increase Waiting Room
case 4	i+1	i-1	i	No Adjustment
case 5		i-1, i+1	i	Increase Weighing Room
case 6		i-1	i, i+1	No Adjustment
case 7	i+1		i-1, i	No Adjustment
case 8		i+1	i-1, i	Increase Weighing Room
case 9			i-1, i, i+1	No Adjustment

referring to likely scenarios that could have occurred and are not guaranteeing such scenarios.) Hence, we conjecture that block i was evicted before being referenced because the Waiting Room was too small. So, we increase the Waiting Room size.

Now consider cases 5 and 8. This is the opposite of the previous situation. The fact that block $i+1$ is in the Waiting Room tells us that block i was in the Weighing Room. This means that the Weighing Room had to evict a block that was to be referenced soon in the future, meaning that the Weighing Room was too small. Hence, adjustments to increase the Weighing Room size is made.

Let us now consider case 2. Case 2 satisfies both of the two previous scenarios, that is, block $i-1$ is found in the Weighing Room and block $i+1$ is found in the Waiting Room. Note, however, that the implications are different. While having block $i-1$ in the Weighing Room only suggests that block i could have been evicted from the Waiting Room, having block $i+1$ in the Waiting Room tells us that block i must have been in the Weighing Room when it was evicted. Hence, the Weighing Room is increased in this situation.

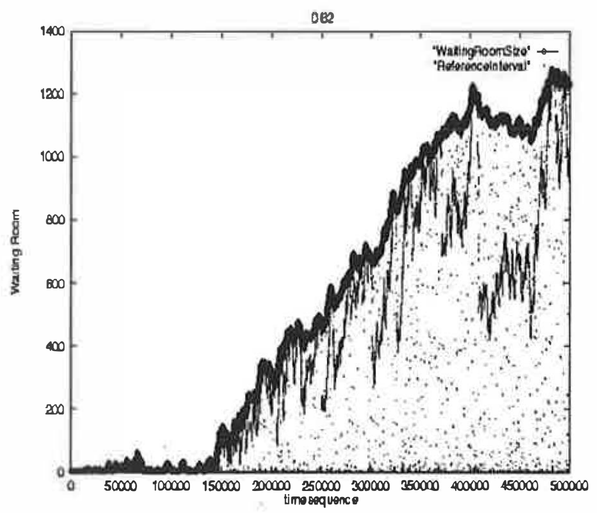
For cases 4, 6, 7, and 9, no solid relation can be deduced. For example, take case 4. The fact that block $i+1$ is in the Weighing Room suggests that block $i+2$ could be in the Waiting Room, but nothing in relation to block i can be deduced. Likewise, the fact that block $i-1$ is in the Waiting Room suggests that block $i-2$ may still be in the Weighing Room, but again, nothing in relation to block i may be deduced. Hence, for these cases no adjustments are made.

Based on these situations and their adjustments, the SA-W^2R scheme adjusts the partitioning of the buffer cache between the Weighing and Waiting Rooms.

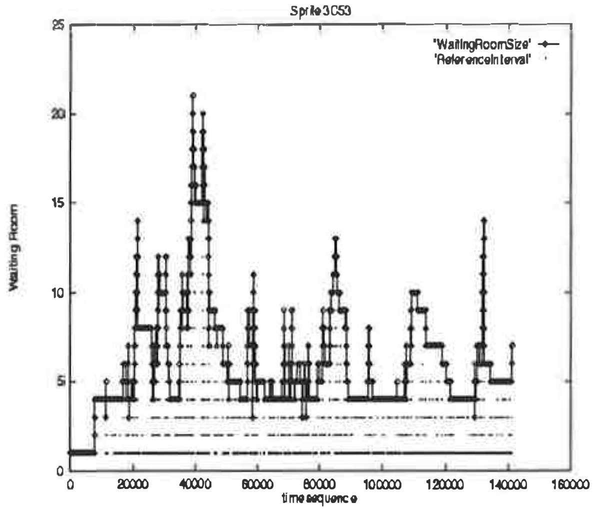
3.3 Adaptability of the SA-W^2R Scheme

Figure 5 shows how the SA-W^2R adapts to the changing workload of the system when the cache size is 3000 for the DB2 and Sprite 3C53 traces that will be explained in the next section. The dark line that looks like the upper boundary of the figure shows the Waiting Room size at each time point. Each dot within the "boundary" represents the access point of a block, that is, the reference interval within the Waiting Room at each time point. For the DB2 trace, the streaky lines going up within the boundary is showing that the reference interval is increasing, while the streaky lines going down shows that the reference interval is decreasing. The figure shows that SA-W^2R is adjusting the Waiting Room as needed.

Figure 5(b) is actually more interesting. Note that the size of the Waiting Room is much smaller compared to the DB2 trace. This is because there is much more sequentiality in this trace. When references are sequential there is no need to increase the Waiting Room. In fact, if the workload is totally sequential, a Waiting Room size of one is sufficient. Hence, for this trace, the SA-W^2R scheme is keeping the Waiting Room size small. The seemingly horizontal lines in the figure show that a majority of the blocks are being referenced at a particular point in the Waiting Room, that is, the reference interval is constant. The lowest horizontal line represents total sequentiality, while horizontal lines above this line show that the reference interval grows, but remains constant over time. The Waiting Room size is adjusted to reflect this change.



(a) DB2



(b) Sprite 3C53

Figure 5: Adaptability of the SA-W^2R scheme for DB2 and Sprite 3C53 traces when the cache size is 3000.

4 Simulation Experiments

In this section, we discuss the trace driven simulation experiments conducted to evaluate the SA-W^2R scheme. A description of the traces that were used is given in the next subsection. In the subsequent subsection, we report and discuss the results of these experiments.

4.1 The Simulator and Traces

The simulator developed to evaluate the schemes is programmed in C++. The basic component in this simulator is the buffer cache module which takes the traces as input. The buffer cache module checks if the block number is in the buffer. If it is a hit, appropriate action, which is dependent on the policy used, is taken. Otherwise, a block fetch request action to the disk is emulated. The block size, size of the buffer, and the policy used for managing the buffer are controllable parameters.

A wide range of traces are used to drive the simulator to show the robustness of the SA-W^2R scheme. Specifically, database traces, the Sprite traces, traces of real application programs, and synthetic traces that show Zipfian distribution are used. Detailed descriptions of these traces are given below.

Database traces: Two traces, namely, DB2 and OLTP, obtained from database systems were used. These traces are identical to the traces used in the papers by Johnson and Shasha [6] and by O'Neil and others [11]. The DB2 trace is obtained from running a DB2 commercial application and contains 500,000 block requests to 75,514 distinct blocks. Obtained from the On-Line Transaction Processing System, the OLTP trace contains records of block requests to a CODASYL database for a window of one hour. It contains a total of 914,145 requests to 186,880 distinct blocks.

Sprite traces: Sprite traces are traces obtained from 4 file servers and 40 clients running the Sprite distributed file system [12]. This file system environment had roughly 30 users who were consistent users with an additional 40 some users who used the system occasionally. Traces were obtained for eight separate periods of 24 or 48 hours. These traces are considered to represent scientific workloads as most of the users were operating system researchers, computer architecture researchers, VLSI circuit designers, parallel processing researchers, etc. Of these traces, the traces that are used in our experiments are traces taken on the 2nd and 3rd periods. The trace that we denote as Sprite 2C39, which is client 39 of the 2nd period, consists of a total of 141,233 block accesses to 19,990 distinct blocks, while the trace that we refer to as Sprite 3C53, which is client 53 of the 3rd period, consists of a total of 239,748 block accesses to 49,277 distinct blocks. Though these two traces were taken from the same system, they themselves are quite different in their characteristics. According to Baker and others [1], the traces in the 2nd period represent general scientific access patterns, while for the 3rd period the workload consisted largely of accesses to very large files making it different from the general workload of the 2nd period.

Application traces: These set of traces, obtained from executing real application programs, are those used in a previous study [3]. The length of these traces are very short compared to the database and Sprite traces ranging from roughly four thousand to thirty-five thousand block accesses. Specific details regarding the characteristics of these applications are given below.

cpp: Cpp is the GNU C-compatible compiler preprocessor. The kernel source was used as input with the size of header files and C-source files of about 1MB and 10MB, respectively.

link: Link is the Unix link-editor. This application is used to build the FreeBSD kernel from about 2.5MB of object files.

ld: Ld is the trace of a linking editor. It has random accesses for both reads and writes. There is no reuse of data, but since the size of a read request is not always 8K, there are occasional reuses at the block level. (A block is 8K bytes.)

XDataSlice: XDataSlice is obtained from a 3D volume rendering software working on a $220 \times 220 \times 220$ data volume, rendering 22 slices with stride 10, along the X axis, then Y axis, then Z axis. This trace accesses blocks in a file with regular strides. There is no reuse of data when rendering along one axis, but moderate reuse is done between rendering along different axes.

Zipfian traces: The Zipfian traces are synthetic traces correspondent with a Zipfian distribution of reference frequencies. The Zipfian distribution of reference frequencies is where the probability for referencing a page with block number less than or equal to i is $(\frac{i}{N})^{\frac{\log a}{\log b}}$ with constants a and b between 0 and 1. The notion of constants a, b, and N is that fraction a of the references accesses fraction b of the N blocks. We generated two types of distributions referred to as ZipfianA and ZipfianB. The ZipfianA trace contains 500,000 references to 75,514 distinct blocks with $a = 0.8$, $b = 0.2$ and $a = 0.7$, $b = 0.3$. ZipfianB contains 914,145 requests to 186,880 distinct blocks with constants a and b the same as that of the ZipfianA trace. The Zipfian distribution and their respective constants were taken as they are known to be a good representation of database reference patterns [11]. The number of requests and distinct blocks were selected to be the same as the DB2 and OLTP traces, respectively.

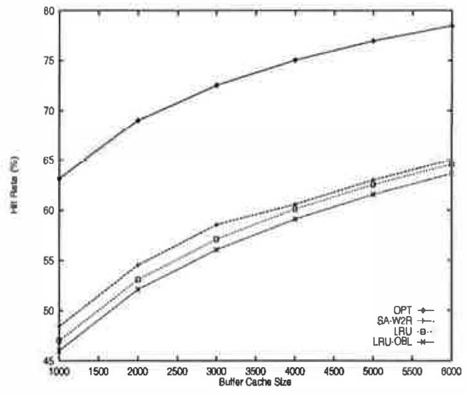
4.2 Results

Figure 6 shows the hit rates for the synthetic workload that shows Zipfian distribution. These workloads are interesting because no sequentiality is found, and we believe this represents one extreme end of reference characteristics. The results using the ZipfianA trace, shown in Figure 6, show that the LRU-OBL scheme is certainly not the choice. (The results are similar for the ZipfianB traces.) It performs even worse than the traditional LRU replacement policy. SA-W^2R shows consistently better performance than both the LRU and LRU-OBL.

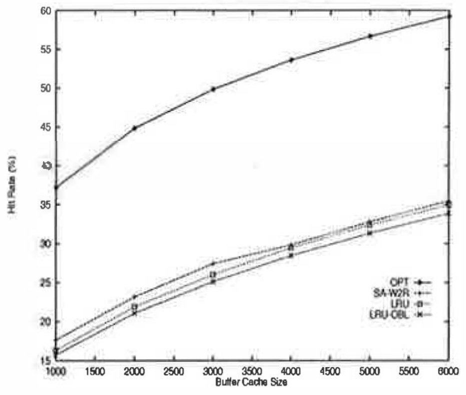
Now, also consider how the schemes deal with purely sequential references, which is the other extreme end of reference characteristics. The LRU replacement policy will incur misses on every reference to a new block resulting in zero hit rate, while for both the LRU-OBL and the SA-W^2R schemes, the hit rate will approach 100% as every logical next block will be prefetched. The results of the two extreme reference characteristics show that the SA-W^2R is a versatile scheme.

Figures 7, 8, and 9 show the hit rates for the SA-W^2R scheme compared with other schemes for real workloads. Regarding the figures, first note that the scales are all different. Also, for all these figures that do not show the hit rates for LRU and/or OPT (the optimal replacement) policies, they are not shown because their margin of difference from the LRU-OBL and SA-W^2R is so large that it makes the lines indecipherable. Hence, we omitted the LRU and/or OPT lines for these cases.

Overall, the performance of the SA-W^2R scheme performs better than all the others. An interesting observation of these results is that the LRU-OBL scheme is superior to the OPT policy. Except for the extreme case where there is only minimum or no sequentiality, the LRU-OBL policy is a good general scheme that could be used for general buffer cache management, and not just limited to sequential reference patterns. Of course, one has to consider how the hit rate performance measure translates to other measures such as response time in real systems. Extra queueing delays incurred by prefetching may limit the performance benefits observable by the user. However, with the advent of RAID

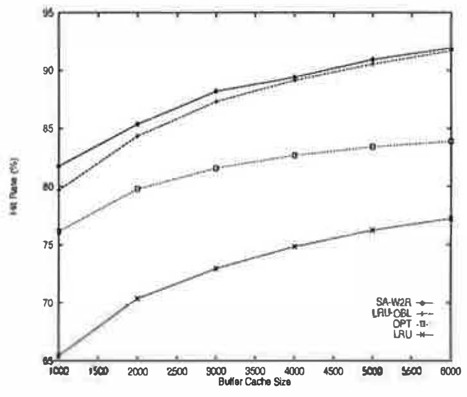


(a) ZipfianA 80_20 distribution

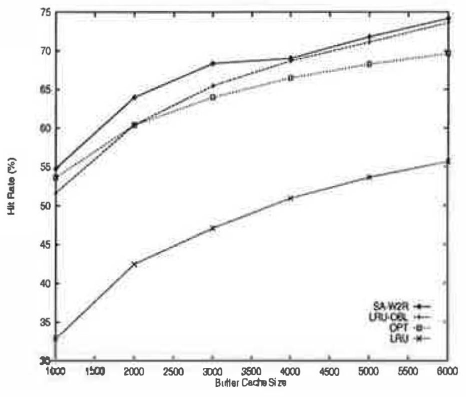


(b) ZipfianA 70_30 distribution

Figure 6: The hit rates of the SA-W^2R scheme for the ZipfianA traces.

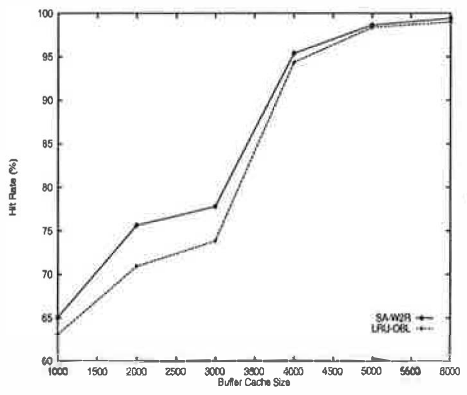


(a) DB2

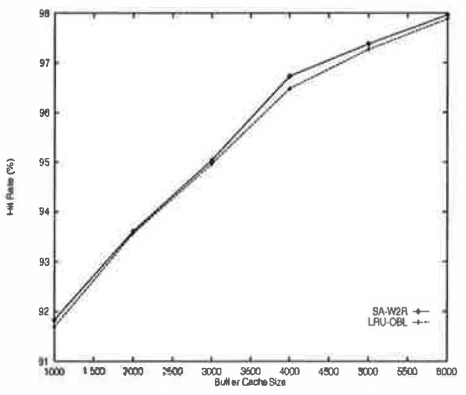


(b) OLTP

Figure 7: The hit rates of the SA-W^2R scheme for the DB2 and OLTP traces.

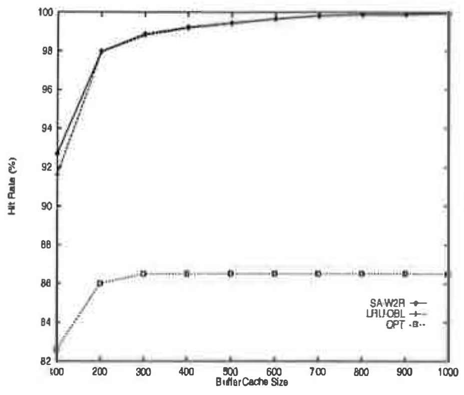


(a) Sprite 2C39

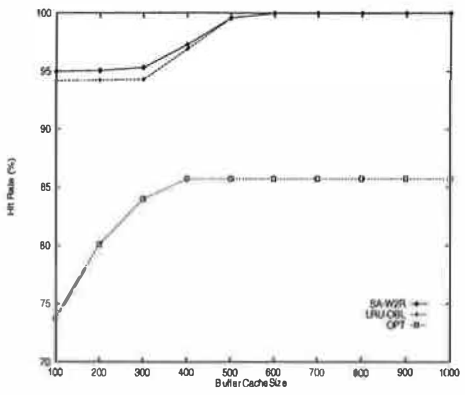


(b) Sprite 3C53

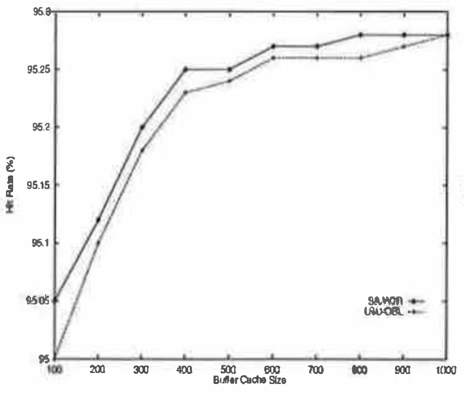
Figure 8: The hit rates of the SA-W^2R scheme for the Sprite traces.



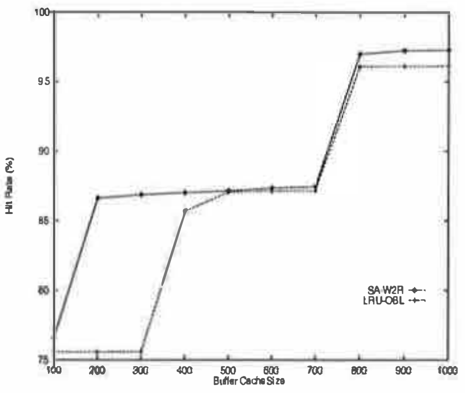
(a) cpp



(b) link



(c) ld



(d) XDataSlice

Figure 9: The hit rates of SA-W^2R scheme for real application traces.

systems and better caching techniques, delays due to this type of queueing should not have aggravated influence on the performance. Hence, hit rates should be a good reflection of the actual performance seen by the user for these schemes.

The SA-W^2R is an even better scheme than the LRU-OBL performing consistently better than the LRU-OBL for all situations including the extreme cases mentioned previously. The maximum performance difference comes from the XDataSlice application where the SA-W^2R has a hit rate that is over 11 percentage points higher than the LRU-OBL scheme. Overall, the performance improvement range somewhere around the 1 to 4 percentage point increase compared to the LRU-OBL.

5 Implementation Experiments

The SA-W^2R scheme was implemented on the GNU/Linux kernel version 2.2.14 on a Pentium III 430 MHz PC with 128 MB of memory. (*At the time of this implementation, the kernel version 2.2.14 was the latest stable version.*) For performance comparison purposes, we also implemented the LRU-OBL scheme. The applications used to evaluate the performance are as follows.

gcc: Compile the GNU/Linux kernel version 2.2.14.

cp: Copy the whole GNU/Linux source code from /usr/src/linux directory to another directory.

tar: Create/extract the tar file for the GNU/Linux source code.

gzip: Compress/uncompress the tar file of the GNU/Linux source code.

grep: Grep the string "linux" from the /usr/src/linux directory.

sort: Sort 1,000,000 random data items.

5.1 Individual Application Performance

Table 2 shows the execution time of each application. The results shown in the table are averages of three executions of each application. Before each execution of each application, the system was rebooted to eliminate the effect of caching from the previous execution.

The results show that the SA-W^2R scheme shows the best performance compared to the original GNU/Linux and LRU-OBL schemes. Specifically, the performance improvements due to the proposed scheme range between 5 to 23 percent compared to the original GNU/Linux scheme.

Note also that the LRU-OBL scheme always performs considerably better than the original GNU/Linux scheme, though worse than the SA-W^2R. This is because regular reference patterns such as sequential references is a dominant characteristic of many of these applications. Hence, it may be argued that this set of applications, specifically, this characteristic, unfairly favors the LRU-OBL and SA-W^2R schemes. To show that the SA-W^2R scheme is a robust solution, we executed with each application a 'random' process that randomly references blocks such that it disrupts regular reference sequences such as sequential block references. Those results are shown in Table 3.

The results in Table 3 show that when regular reference behavior is disrupted LRU-OBL may perform worse than the original GNU/Linux management scheme. It also shows, however, that the SA-W^2R scheme consistently performs better though its improvement is now somewhat smaller. This shows that the SA-W^2R is quite robust in its management of the buffer.

5.2 Concurrent Execution of Multiple Applications

Using the same methodology for the experiments, we measured the performance of applications when multiple applications were executed concurrently. Table 4 shows the applications that were executed concurrently (Groups 1 to 3) and their respective execution times using the different buffer management schemes.

As the concurrently executing applications influence the buffer and CPU resource allocation, the execution times of the applications increase considerably. Again, in all of the situations, the SA-W^2R scheme shows the best performance. The improvements range from negligible (approximately 1% for gzip (compress) of Group 2) to an approximately 20% reduction (for the gzip (uncompress) of Group 3) in execution time.

To measure the overhead of the bookkeeping and management of information for adjusting the room sizes, we added a CPU bound process that continuously does simple add operations to each of the groups of applications. The results of these experiments are shown in Table 5. Note that the effect of the CPU bound process on the execution of each application depends on the characteristics of the applications that are executing. For applications of Group 1, the increase in the execution time is small, while for those of Group 2, the increase is substantial. (This can be observed by comparing the execution times of Tables 4 and 5.) Note though, that the increase in execution time of the CPU bound process (for Groups 1 and 3) are quite small, implying that the overhead for maintaining relevant information for dynamic room partitioning is quite small.

6 Conclusion and Future Works

In this paper, we proposed the SA-W^2R scheme that is in line with the LRU-OBL scheme, that is, a simple and practical scheme that integrates prefetching and replacement policies. It is simple and practical, and yet it is modular in that any replacement policy deemed appropriate may be incorporated into the scheme. Simplicity results in a scheme that is easy to implement, hence practical.

An extensive implementation study was done, and experimental results show that the SA-W^2R shows better performance compared to the original GNU/Linux and LRU-OBL implementations.

Issues such as quantifying the benefits of the modularity of the Weighing Room or the effect of hint-based prefetching on the SA-W^2R scheme are being considered. Performance of prefetching is also strongly influenced by the performance of the disk system as well, and, thus, their interaction must be studied more closely.

Table 2: Average execution time for applications using different buffer management schemes (in seconds).

Applications	Original Linux	LRU-OBL	SA-W^2R
gcc	244	241	230
cp	59.40	53.31	48.71
tar (create)	61.02	59.87	55.31
tar (extract)	41.70	39.93	36.26
gzip (compress)	72.87	64.73	56.11
gzip (uncompress)	21.45	19.99	17.23
sort	47.32	45.14	42.57
grep	46.92	38.28	37.01

Table 3: Average execution time for applications with a 'random' process disrupting regular reference behavior using different buffer management schemes (in seconds).

Applications	Original Linux	LRU-OBL	SA-W^2R
gcc	394.15	393	387.24
cp	302.32	304.64	297.57
tar (create)	312	304.89	298.99
tar (extract)	46.55	49.22	43.58
gzip (compress)	76.93	69.17	66.12
gzip (uncompress)	22.47	22.34	20.84
sort	54.55	55.29	54.25
grep	294.08	277.59	272.65

Table 4: Average execution time for groups of applications executed concurrently (in seconds).

	Applications	Original Linux	LRU-OBL	SA-W^2R
Group 1	cp	325.73	318.44	313.23
	tar (create)	329.57	323.79	319.53
Group 2	gzip (compress)	95.67	98.26	94.35
	sort	91.80	92.06	88.56
Group 3	tar (extract)	77.96	83	75.43
	gzip (uncompress)	59.70	50.91	47.54
	grep	130.42	127.42	122.55

Table 5: Average execution time of groups of applications executing concurrently with a CPU bound process (in seconds).

	Applications	Original Linux	LRU-OBL	SA-W^2R
	CPU bound process only	222	222	222
Group 1	cp	329.57	315.82	311.94
	tar (create)	331.95	319.74	314.65
	CPU bound process	235	235	236
Group 2	gzip (compress)	138.77	143.08	137.52
	sort	138.74	141.71	137.17
	CPU bound process	315.07	315.47	314.96
Group 3	tar (extract)	90.8	91.58	85.26
	gzip (uncompress)	79.83	61.32	58.01
	grep	146.02	138.91	129.57
	CPU bound process	246.01	247.51	248.1

7 Acknowledgement

The authors would like to thank Gerhard Weikum and Theodore Johnson for providing us with the DB2 and OLTP traces and Jongmoo Choi for helping with the application traces. We would also like to thank our shepherd, Stephen C. Tweedie, for helping us finalize this paper.

References

[1] Mary G. Baker, John H. Hartman, Michael D. Kupfer, KenW. Shirriff, and John K. Ousterhout. Measurements of a Distributed File System. In *Proceedings of the 13th ACM SOSP*, pages 198–212, Pacific Grove, CA, October 1991.

[2] Pei Cao and Edward W. Felten. Implementation and Performance of Integrated Appplication-Controlled File Caching, Prefetching, and Disk Scheduling. *ACM Transactions on Computer Systems*, 14(4):311–343, November 1996.

[3] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. A study of integrated prefetching and caching strategies. In *Proceedings of 1995 Joint ACM SIGMETRICS and Performance Evaluation Conference*, pages 188–197, 1995.

[4] Fay Chang and Garth A. Gibson. Automatic I/O Hint Generation through Speculative Execution. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation*, pages 1–14, February 1999.

[5] K. Curewitz, P. Krishnan, and J.S. Vitter. Practical Prefetching via Data Compression. In *Proceedings of the 1993 ACM SIGMOD Conference*, pages 257–266, May 1993.

[6] Theodore Johnson and Dennis Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th VLDB Conference*, pages 439–450, 1994.

[7] David Kotz and Carla Schlatter Ellis. Practical Prefetching Techniques for Multiprocessor File Systems. *Journal of Distributed and Parallel Databases*, 1(1):33–51, January 1993.

[8] Donghee Lee, Jongmoo Choi, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. On the Existence of a Spectrum of Policies that Subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) Policies. In *Proceedings of the 1999 ACM SIGMETRICS Conference*, pages 134–143, 1999.

[9] Hui Lei and Dan Duchamp. An Analytical Approach to File Prefetching. In *Proceedings of the 1997 USENIX Annual Technical Conference*, pages 275–288, January 1997.

[10] Todd C. Mowry, Angela K. Demke, and Orran Krieger. Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, October 1996.

[11] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *Proceedings of the 1993 ACM SIGMOD Conference*, pages 297–306, May 1993.

[12] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch. The Sprite Network Operating System. *IEEE Computer*, 21(2):23–36, February 1988.

[13] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed Prefetching and Caching. In *Proceedings of the 15th ACM SOSP*, pages 79–95, December 1995.

[14] J. T. Robinson and N. V. Devarakonda. Data Cache Management Using Frequency-Based Replcement. In *Proceedings of the 1990 ACM SIGMETRICS Conference*, pages 134–142, 1990.

[15] Alan Jay Smith. Sequential program prefetching in memory heirarchies. *IEEE Computer*, 3(3):7–21, December 1978.

[16] Alan Jay Smith. Disk cache-miss ratio analysis and design considerations. *ACM Transactions on Computer Systems*, 3(3):161–203, August 1985.

[17] Andrew Tomkins, R. Hugo Patterson, and Garth A. Gibson. Informed Multi-Process Prefetching and Caching. In *Proceedings of the 1997 ACM SIGMETRICS Conference*, pages 100–114, June 1997.

Translucent Windows in X

Keith Packard
XFree86 Core Team, SuSE Inc.
keithp@suse.com

Abstract

The X Window System allows multiple windows to occupy the same coordinates on the screen. The core protocol defines which portions of each window are visible and which are occluded by overlapping windows, but the overlapping windows are always completely opaque.

Various techniques can be used to simulate non-opaque windows in controlled environments. The Shape Extension can be used to make areas of the window transparent. A background of "None" can be used to inherit the contents of the screen in the region occupied by the window when it is first mapped. Where available, hardware overlays can be used which expose a transparent pixel value.

None of these techniques can be used for translucency in a general way; hardware overlays and the Shape Extension can only provide transparency and cannot blend the pixel colors together. A background of "None" cannot be used when the occluding windows are to be reconfigured or when the occluded region contents are expected to change.

The X Translucent Window Extension is described which solves the general translucency problem by assigning alpha values for pixels in occluding windows. These values are used to blend the occluding window contents with the occluded region for display. The details of managing translucent window hierarchies, reparenting translucent windows and X visual differences between blended pixels are discussed.

1 Translucency

Physically, translucent objects absorb some, but not all, of the light passing through them. The color of the object affects which wavelengths are most strongly absorbed. Visually, translucent objects appear to affect the color and brightness of objects beyond them.

The effect of light on a translucent object can be simulated by blending the color of the translucent object with that of objects beyond it. When dealing with computer images, translucency can be described as a mathematical operation on the color data of a collection of images. This mathematical composition of images was formally defined by Thomas Porter and Tom Duff in 1984 [PD84]. They define formulae which use color data in conjunction with a per-pixel opacity value called "alpha". With these formulae many intuitive image manipulations can be performed.

1.1 Image Compositing Operators

Each of the operators defined by Porter and Duff operate independently on each of the color channels in each pixel. The equations are abbreviated to show the operation on a single channel of a single pixel.

A common compositing operation is to place one image over another. Transparent areas of the overlying image allow the underlying image to show through. Opaque areas hide the underlying image while translucent areas blend the two images together. By defining the "alpha" of a pixel as a number from 0 to 1 measuring its opacity, a simple equation combines two pixel colors together:

$$C_{result} = C_{under} \cdot (1 - \alpha_{over}) + C_{over} \cdot \alpha_{over}$$

Porter and Duff call this the "over" operator.

Another common operation is to mask an image with another; transparent areas in the mask are removed from the image while opaque areas of the mask leave the image visible.

$$C_{result} = \alpha_{mask} \cdot C_{image}$$

This is the "in" operator. They provide a complete compositing algebra including other operations; only these two are needed for this extension.

One important aspect of this model is that it creates a new image description which attaches another value "alpha" to each pixel. This value measures the "opacity" of the pixel and can be operated on by the rendering functions along with the color components.

1.2 Destination Alpha

Sometimes it is useful to create composite images which are themselves translucent, in other words, contain alpha values. This effect can be achieved by augmenting the operators with an operation which produces a composite alpha value along with the color values. For the "over" operator, the composite alpha value is defined as:

$$\alpha_{result} = \alpha_{under} \cdot (1 - \alpha_{result}) + \alpha_{over}$$

The "in" operator composite alpha value is:

$$\alpha_{result} = \alpha_{mask} \cdot \alpha_{image}$$

The resulting images can now be used in additional rendering operations.

1.3 Premultiplied Alpha

Visible in the above equations for computing the "over" operator is the asymmetry in the computation of alpha and the color components:

$$\alpha_{result} = \alpha_{under} \cdot (1 - \alpha_{over}) + \alpha_{over}$$

$$C_{result} = C_{under} \cdot (1 - \alpha_{over}) + C_{over} \cdot \alpha_{over}$$

This is "fixed" by respecifying the image data as being "premultiplied by alpha". Each color component in the image is replaced by that component multiplied by the associated alpha value. Blinn [Bli94] notes that premultiplied images easily provide the correct results when run through long sequences of operations, while non-premultiplied images involve awkward computations.

2 Uses For Translucent Windows

As user interface design moves forward with increasing graphics performance, things formerly passed over as computationally intractable are now quite reasonable. Here are a couple of ideas how translucent windows could be used.

2.1 Window Management Effects

It is possible to simulate translucency during some window manipulation operations to provide additional feedback for the user. This can be implemented by capturing a static image of the window and the desktop, blending them, and updating the display.

This means that any changes that occur to the window contents during the operation cannot be reflected dynamically on the screen, limiting this to window movement operations. Moving the blending operations into the window system allows for the dynamic composition of application and underlying images.

Applications should not be required to cooperate to provide these effects. This requires an external control over window hierarchy translucence.

2.2 Transient Data

User interface elements which appear transiently over application windows such as menus and dialog boxes often occlude information useful in the operation of the transient action. By making the background of the transient window transparent and using translucency to highlight dialog elements, an effective user interface element can be usable and yet still allow interpretation of occluded application data.

While this can be effected by eliminating windowing for the transient elements and rendering them directly to the application window, the ability to continue to use window management and other windowing metaphors for these elements provides a strong incentive to incorporate these semantics into the window system.

2.3 Overlays and Annotation

Applications with complex image displays frequently present the ability for users to overlay associated information or annotate the image without affecting the underlying image data. These needs are satisfied today only with graphics hardware that supports overlays. But even where supported, the annotations are usually limited to fewer color planes than the main image and cannot incorporate translucence in the image, only opacity and transparency.

Again, these effects can be implemented by the application, however, the underlying graphics are often expensive to redisplay. Moving these operations into the window system provides the ability to use overlay hardware where available and gracefully fallback to software when necessary.

Both transient windows and annotations require per-pixel translucency that tracks rendering operations.

3 The X Rendering Extension

The X Window System [SG92] is a networked, extensible window system providing hierarchical windows. It was designed to operate on a wide variety of graphics hardware, from simple frame buffers to high-end graphics systems with overlays, underlays, accumulation buffers, z-buffers, double buffering, etc. Over the last 12 years, it has become the standard window system for Unix and Unix-like systems.

The X Rendering Extension [Pac00] provides a new rendering architecture for X applications including image compositing, sub-pixel positioned geometric primitives and application management of glyphs. The X Translucent Window Extension builds on the image composition ideas and mechanisms provided by the X Rendering Extension.

3.1 New Objects

The "PictFormat" object holds information needed to translate X pixel values into color and alpha data. For TrueColor visuals, the color data are extracted directly from the pixel while pseudo color visuals use a separate Colormap. The PictFormat references the appropriate Colormap in that case. It also indicates the portion of the pixel which contains alpha information (if any).

To encapsulate rendering state and color information, X Drawables (pixmaps and windows) are wrapped inside a new "Picture" object. An external pixmap Picture containing alpha data can be associated with the Picture. This external alpha data overrides any embedded alpha data.

3.2 Rendering Operators

The X Rendering Extension uses a modified version of the Plan 9 rendering primitive [Pik00] as the basis for image composition:

$$C_{result} = (C_{image} \text{ IN } C_{mask}) \text{ OP } C_{result}$$

In the Plan 9 window system, OP is always OVER. The extension allows any of the operators defined by Porter and Duff along with a special operator designed for drawing anti-aliased graphics adapted from OpenGL.

Using this basic rendering primitive, the extension defines geometric operations by specifying the construction of an implicit mask which is then used in the general primitive above. Anti-aliased graphics can be simulated by generating implicit masks with partial opacity along the edges.

3.3 Color Management

The core X protocol defines all rendering primitives in terms of pixel values. While this works when the rendering is done with boolean operations, it's color-based rendering must have a color interpretation for each pixel value.

The PictFormat object contains the information necessary to translate a pixel value into a color. The converted color can then be composited with other colors to generate the displayed color. Once a final color is computed, the extension converts it back to a pixel value using a fixed palette with optional dithering to improve color fidelity. Pseudo color displays use a fixed palette generated by the server for all images; the flexibility of a dynamic palette was discarded in favor of reduced colormap flashing and simpler code.

4 Windowing Semantics

X provides a hierarchical window system. Windows provide a view onto a document or scene. Windows are stacked over or under their peers and contain subwindows. The top of the window hierarchy is called the root and is distinguished from other windows by being contained in no window.

The visible area of a subwindow is confined to the visible area of the containing window. The visible area of a window is occluded by all subwindows. A window along with all subwindows stack together with respect to that windows peers.

Given the above semantics, the visible portion of a window can be found by computing the portion of the window within the visible portion of its including window and subtracting the areas of any subwindows and overlying peer windows. The visible portions of each window are combined to form the final displayed image. As the visible portions of each window form a partition of the root window area, each pixel on the screen belongs to the visible portion of precisely one window.

4.1 Transparent Windows

Graphics hardware frequently provides the ability to "color-key" one frame buffer over another frame buffer. This compositing is done in hardware, and allows either the underlay or overlay to be display, but not a combination of the two.

For hardware with this capability, X lists pairs of visuals for each screen, one as the underlay and another as the overlay. Transparent areas in windows created in the overlay visual show through contents of windows in the underlay. The transparent areas are typically specified with a special pixel value.

The semantics of this mechanism are meant to expose the underlying hardware abilities, rather than match the windowing model and can generate surprising results. One such surprise was that transparent pixels in the overlay visual would unconditionally show through to the nearest occluded window in the underlay, even if intervening windows existed in the overlay.

The X Shape Extension [Pac89] provides a mechanism for altering the visible region of a window. This affects graphical output as well as pointer input: areas outside of the shape do not receive pointer events. The Shape Extension can be used to implement partial window transparency, but the effect on input is usually not desirable. Additionally, the regular X semantics fail to ensure that the occluded window contents will be preserved by the X server for redisplay. A large class of applications fail to perform acceptably when their contents are damaged, making shaped windows unsuitable.

The final problem is that the Shape Extension is implemented by modifying the window clipping regions within the server. These regions are represented as lists of rectangles. Complex bitmap shapes generate clip lists of thousands of rectangles, slowing the server unacceptably.

4.2 Windowing as Compositing

An informal description of typical window system semantics is that of overlapping pieces of paper on a desktop. This characterizes the original intent as developed at Xerox, but seems far removed from the formal X semantics described above.

Reinterpreting those in terms of image composition provides a more transparent description.

A window is composed of an image and zero or more stacked subwindows. The visible image of a window is generated by composing the window with the visible image of its subwindows using the *over* operator. The alpha value of a window is 1 inside the shape of the window and 0 outside.

5 Translucent Windowing Semantics

Existing systems provide two interpretations for translucency. A reasonable system will provide mechanisms for both and also allow hardware acceleration when possible.

Systems based on hardware overlays provide a special pixel value that exposes data in the underlying window. These embed transparency in the pixel value itself. Each pixel can be either opaque or transparent, but not translucent.

Systems designed to animate window operations provide an external opacity value that controls the blending of a window to the desktop. The data within the window needn't contain opacity information, rather that is applied by the external window management agent to affect the display of the resulting image.

6 Prototype

To provide a framework for exploring window compositing, a simple prototype was constructed that allows for arbitrary compositing between windows.

As seen above, windowing can be described in terms of compositing images in layers. The displayed image of a window is formed by compositing the window image data along with the displayed images of each inferior window.

This suggests a relatively straightforward, if somewhat inefficient, implementation of windowing. Instead of providing window layering by clipping each rendering operation to a single shared image, window layering can be implemented by compositing separately rendered images.

6.1 Prototype Implementation

The image data for each window is kept in an off-screen image buffer. The complete displayed image is formed by recursively compositing these images together.

A separate displayed image buffer is used to render the composite image of the window and subwindows. The window image data are copied to that displayed image buffer. Finally, the displayed image for each inferior is generated and composited to the displayed image buffer. The root window uses the frame buffer itself for the displayed image.

As all rendering operations now occur off screen, the X server must update the displayed image for the root window whenever visible changes occur. The prototype keeps a single region which encloses any damage. Each rendering command updates that region. When the server is about to wait for additional X requests, it recurses through the window hierarchy updating the damaged areas of each displayed image. Finally, the damaged region is emptied.

6.2 Prototype Results

The initial prototype provides an alternate implementation of X windowing. By itself, that is not terribly useful. To demonstrate the capabilities of the architecture, the server was modified to mask each "override redirect" window with a constant alpha value of 2/3. This makes most menus appear translucent.

The prototype is minimally functional. Performance is poor, enough for a demonstration but not for real applications. Nonetheless, it is interesting to see the effect of translucency on real applications and gauge the usability of various user interface ideas. For example, making menus entirely translucent leads to readability problems. The text should probably be opaque and outlined in a contrasting color while the background of the menu remains translucent. The Phillips TiVo, a Linux-based video storage device, displays text in this manner.

The prototype can be extended to support the Translucent Window Extension; adding semantics for different compositing operators is quite easy once the entire contents of every window is available.

7 Improving the Design

While the prototype demonstrates an easy intuitive architecture for window compositing, it uses memory for window image data which is not used to generate the final display. It also recomposites window images at each level of the hierarchy, making deep window trees perform poorly. The prototype has been useful, but architectural changes are needed for a production system.

The final design should be equivalent to the existing X windowing system in the absence of translucency.

7.1 Opaque Windows

Windows with a constant alpha value of 1 (those without an alpha channel or mask) can be directly rendered to the enclosing window's image buffer. This is a generalization of the standard X rendering model in which all windows share the frame buffer for image data. Similar rules apply here: the enclosing window and any occluded windows must clip rendering out of that area. When compositing the displayed image, any occluded areas from other windows must not touch the occluding pixels.

7.2 Occluded Window Regions

Sections of windows occluded by opaque windows are not needed to generate the final displayed image on the screen and so need not be contained in any image. Existing X clipping operations can be used to modify rendering operations in this case.

For the root window, this will allow direct rendering to the frame buffer, entirely avoiding the cost of compositing windows while ensuring acceleration of rendering operations with any display hardware.

7.3 Translucent Subwindows

The prototype stores the entire window image off screen so that the image can be composited with subwindows. Instead, only the portion of a window covered by translucent windows need be stored in a separate buffer with the remainder rendered directly to the displayed image buffer.

This gives each window two regions, one containing the area rendered directly to the displayed image buffer and a second containing the area covered by translucent subwindows which must be rendered to an off-screen image buffer.

7.4 Multiple Frame Buffers

The prototype works only for a homogeneous display; all windows must be true color at the same depth. To extend this for hardware with overlays, the implementation must allow for windows of different depths and visual classes.

For opaque windows, providing separate displayed image buffers where needed in the hierarchy is sufficient. Translucent windows must be blended together to be displayed. For windows not using a true color visual, the pixel values must be converted to color values, blended together then converted back to pixel values and stored in the displayed image buffer of an appropriate format. The semantics for this conversion are described in detail by the Rendering Extension.

8 X Translucent Window Extension

The X Translucent Window Extension exposes semantics for window translucency to applications, allowing them to manipulate the composition of window data to the screen. The Translucent Window Extension uses the Rendering Extension compositing primitive with a fixed OVER operator.

$$C_{result} = (C_{image} \text{ IN } C_{mask}) \text{ OVER } C_{result}$$

This single operator combines the masking of the "in" operator with the blending of the "over" operator. The Plan 9 window system uses this primitive for all graphics operations. Blinn suggests that for image composition, the OVER operator is sufficient for nearly every operation. This extended primitive incorporates the ability to render geometric objects, text and images with external alpha channels with a single simple operation.

8.1 Embedded Alpha Values

To allow applications to control opacity while rendering, pixel values must map to opacity values. Overlay hardware frequently uses special pixel values, but is usually limited to either transparent or opaque pixels. A more general solution associates a deeper alpha value with each pixel allowing it to be rendered along with the pixel values.

The Rendering Extension describes pixel formats including alpha; those resulting alpha values may be directly used as window opacity values when that is appropriate. For visuals without embedded alpha values, an external pixmap will contain the alpha values. For windows without embedded or separate alpha values, the extension uses a constant alpha value of 1.

8.2 External Opacity Control

This operations is frequently used to take an existing window and blend it over the screen. Menus, cursors and dialogs are examples where access to underlying information is desired even when the dialog may occlude that information. The essential requirement is for a separate alpha channel that can cause the window contents to go from opaque to transparent without changing the contents of the window. The "mask" operand in the Plan 9 primitive provides such control.

The Rendering Extension allows that mask to be "tiled" over the operation, repeating the mask in both directions to cover the area. By creating a 1x1 mask, the resulting single value controls the blending of the entire window.

9 Directions

There are several lines of work related to this extension. While the X Rendering Extension has a preliminary specification, there remains significant implementation to be completed which may affect that specification.

The server windowing infrastructure needs to be modified to support translucent windows. There are several other potential enhancements to the X Window System which would be able to take advantage of these architectural changes. Among them are an improved save-under system, a mechanism for changing the hardware accelerated visuals on each screen, changes to the backing store system and translucent multi-color cursors.

Finally, the Translucent Window Extension protocol must be formally defined and implemented. With the architectural infrastructure in place, this should be a relatively small effort.

10 Conclusion

Image compositing forms the basis of many modern rendering systems, from PostScript and PDF to GL and the Quartz windowing system. Extending the X windowing semantics to allow for window-level compositing provides a powerful new tool for application development.

Acknowledgments

I thank SuSE for encouraging me to work full time on XFree86. I'd also like to thank the ALS organizers, Usenix and VA Linux for supporting the second XFree86 Hothouse.

References

[Bli94] Jim Blinn. Compositing theory. *IEEE Computer Graphics and Applications*, September 1994. Republished in [Bli98].

[Bli98] Jim Blinn. *Jim Blinn's Corner: Dirty Pixels*. Morgan Kaufmann, 1998.

[Pac89] Keith Packard. Shape Extension Protocol, Version 1.0. X consortium standard, X Version 11 Release 6.4, 1989.

[Pac00] Keith Packard. A New Rendering Model for X. In *FREENIX Track, 2000 Usenix Annual Technical Conference*, pages 279–284, San Diego, CA, June 2000. USENIX.

[PD84] Thomas Porter and Tom Duff. Compositing Digital Images. *Computer Graphics*, 18(3):253–259, July 1984.

[Pik00] Rob Pike. *draw - screen graphics*. Bell Laboratories, 2000. Plan 9 Manual Page Entry.

[SG92] Robert W. Scheifler and James Gettys. *X Window System*. Digital Press, third edition, 1992.

Developing Drivers and Extensions for XFree86-4.x

Dirk Hohndel
SuSE Linux AG
Nürnberg, Germany
hohndel@suse.de
Robin Cutshaw
Intercore
Atlanta, GA, USA
robin@intercore.com

Abstract

This paper gives an introduction to driver development for XFree86-4.x. After a quick analysis of the existing problems in the previous XFree86 design, it describes the module loading architecture and the key interfaces that a graphics hardware driver for XFree86 must support.

1 Introduction

XFree86 has been the standard implementation of the X Windows System [1] for PC Unix systems for quite a while now; The XFree86 Project [2] was started in 1992 and incorporated in 1994. From the very beginning XFree86 was an open source project (even though back then the term "open source" wasn't widely used), still the first supported platforms were proprietary commercial Unix implementations. Open source simply came as a natural consequence of extending the X Window System. And with the success of Linux, XFree86 became one of the most often used implementations of the X Window System and to some extend the new driving force behind the development of X11.

Having started soon after the official release of X11R5 in 1991, XFree86 was in its design and source layout based on the original X386 work by Thomas Rll which was donated into X11R5 by SGCS.

This had many implications for the fundamental assumptions upon which this design was based. One of the more important (and more devastating) ones was the fact that there was no real design document. While Jim Gettys et al. in [3] describe the fundamental design of X11, and while Elias Israel and Erik Fortune in [4] explain the fundamental X Server design, there is no document that defines the design behind X386 and, subsequently, behind XFree86 versions 1 through 3.

The device dependent X (ddx) code that handles the programming of the hardware was always an area where companies tried to maintain some intellectual property outside the (open source) sample implementation of X11. This was true both for companies like SGCS as well as for the traditional Unix vendors. Therefore, little progress in the ddx code became part of the sample implementation

While X11R6 changed some of the source layout and some of the structure of the device independent X (dix) code, the design of the ddx was largely untouched. It was based on late-80s / early-90s design decisions that, to make matters worse, were not documented.

So in oder to develop drivers for the older versions of XFree86 one had to sit down and read the code. And we are talking about quite a large amount of code. The xfree86 ddx of XFree86-3.2, for example, contains about 1000 source files and roughly 350000 lines of code. For version 3.3.6 this increased to about 1400 source files with about 520000 lines of code.

Early on David Wexelblat, one of the founders of XFree86 and a member of the XFree86 Core Team provided a sceleton driver [5], but even using that it was still necessary to understand very subtle interdependencies and many assumptions that the overall code made about the type of hardware used, in order to successfully write new code for XFree86.

One of the more involved assumptions was that the graphics board in a PC would be VGA compatible. The code loading color maps, for example, relied on the fact that the default way to do so on a VGA board would work. This and other similar assumptions of course turned out to be problematic as new more advanced architectures for graphics card started to become available.

But aside from the technical details of the driver implementation, there were some logistical consequences to the design as well. The hardware drivers were part of the X server binary. This monolithic architecture of XFree86 before version 4 forced (in most cases, there were exceptions for some types of extensions and for Xinput drivers) the release of a complete server binary if a new driver (or an updated version of a driver) was to be released. At first glance this doesn't sound so bad. But the logistical problem stems from the fact that XFree86 supports more than a dozen OSs, including Linux, FreeBSD, NetBSD, OpenBSD, Solaris, SCO, QNX and even OS/2, to name a few. Several of these are supported on multiple hardware architectures.

Releasing even a simple driver therefore implied to get hold of all these platforms (or at least the more popular ones), to create server binaries and installation instructions, and to provide those to the users. And of course there's the added complexity if more that one group is working on improvements (so if you want the new extension from group A, e.g., VMware's new DGA version, and the new driver from group B, e.g. SuSE's new Rage128 driver, then you had to rely on these two groups talking to each other and importing each other's new features).

2 Module Loading Architecture

With the advent of XFree86 version 4 some fundamental assumptions about extending its functionality thankfully have changed. Metro Link has donated a module loading architecture that XFree86 has enhanced and extended so that now there exists a portable architecture to load object files (and even ar libraries) at run time into the address space of a process.

Several diffent object formats are supported in this architecture. While a `dlopen()` based loader is included, implementation issues with unresolved symbols in modules make it very hard to use. The obvious advantage of a `dlopen()` based loader (full support in system debuggers) contrasts the down side of implementation limitations and the non- portability of the resulting modules.

The most often used file format are ELF objects as created, e.g., on a Linux system. Additionally, a.out and COFF are supported as well. The host operating system and its preferred object file format have no influence on the selection, the loader code is fully selfcontained and needs no further support from the host OS beyond standard libc interfaces.

The server binary (which obviously has to be in the native executable file format of the host OS) can simply load modules at run time by calling

```
LoadModule(ModuleName, Path, SubDirs, Pattern,
           OptionList, ModReq, &errmaj, &errmin)
```

This allows the server to load the given module, finding it according to the rules given in Pattern in the SubDirs of Path. The options given in the OptionList are passed to the setup function of the module when the module is first initialized. The loader framework can ensure that the required ABI versions given in ModReq are met.

Which modules are loaded at run time is defined through compiled in defaults (e.g., a bitmap font renderer is mandatory to be loaded), through the XF86Config file and through dependencies among the modules. The deferred symbol resolution strategy allows to load modules that have dangling references to other modules that will be loaded at a later point in time. Additionally, modules can reference symbols that the main server binary exports to them.

When a module is first loaded, before all symbol references are fulfilled, the loader code searches for a data element named ¡ModuleName¿ModuleData that contains the module version information (including ABI versions that the module supports) and references to a setup and teardown function. In the next step the setup function is called with the options provided during the call to `LoadModule()`.

At this phase only the symbols that are exported from the main server binary and all in-module references are resolved. The setup function therefore needs to avoid calling functions that are provided through other modules.

Once all modules are loaded, the remaining symbols in the modules are resolved and the main program can match addresses to symbols by looking them up with FunctionPtr = LoaderSymbol(FunctionName).

3 Logistical considerations

This modular loading architecture allows to provide just the modified/new driver (or extension) module instead of the full X server. The server can load this new module at run time and utilize its new features. The architecture provides an additional simplification. Since the loader code does not rely on the host OS in order to read and interpret object files, all the OSs on the same hardware architecture can share the same type of modules as well. Releasing a driver for a new card now is as easy as releasing a single ".o" file for the x86 architecture, and maybe the same for other hardware platforms that the device can be used on.

The installation of this new driver implies simply to copy it to a well known location (e.g., /usr/X11R6/lib/modules/drivers for graphics hardware drivers) and, if this is a new driver, updating the XF86Config file accordingly.

The logistical problem now boils down to issues of ABI versions in the loader code, the need for exported symbols from the main server binary or maybe the host system libc implementation, and of course the issue of authentication of modules. Let's quickly go through these one by one.

ABI version The application binary interface that defines the loader architecture in XFree86 has a versioning scheme that allows (through major.minor numbering) compatible and incompatible changes to different parts of the ABI. By defining the ABI version in the XF86ModuleVersionInfo of a module the server can determine if it is able to load and execute the module.

Since different classes of modules have different ABI classes it is possible to change the ABI for one subset of the server without having to modify unrelated modules. For example, video drivers and font renderers have different ABI classes.

exported symbols The XFree86 ANSI C Emulation ABI defines a set of entry points and variables from the main server binary that can be referenced from the module. This includes many of the ANSI libc functions that are provided to the modules by means of special wrapper functions that allow portable access to host libc routines. This is necessary to make driver modules independent from special versions of libc and to make modules protable between different OSs for the same architecture.

authentication of modules Modules contain information on the manufacturer and the version of XFree86 against which they were built. This is additional information for the end user only, the relevant ABI versioning does not use this information. In order to be able to determine whether a module has been tampered with, authentication code that gives a cryptographic signature of the module (which can then be compared against a vendor-provided list of valid signatures) will be added.

At this point the code necessary to do this has not been fleshed out in detail.

4 Writing a Driver

What remains is the technical problem of developing drivers and extensions for XFree86. This still is a very dificult task, but that as well has become easier. After unpacking the XFree86 sources a detailed design document can be found at `xc/programs/Xserver/hw/xfree86/doc/DESIGN` This document covers the important details of the new server design and provides step by step analysis of the flow of control in the server, the mandatory interfaces that a driver needs to provide and the optional driver functions. Furthermore the data structures used in the drivers are defined and explained, handling of bus resources and the helper functions that the server provides for commonly needed tasks in a driver.

While all the details are in this design document, a quick overview of the necessary parts shall be given here.

- a ModuleData variable as described above that contains the necessary information to load and initialize the module

```
static MODULESETUPPROTO(zzzSetup);

XF86ModuleData zzzModuleData =
{ &zzzVersRec, zzzSetup, NULL };
```

  a sample version info would look like this

```
static XF86ModuleVersionInfo zzzVersRec =
{
 "zzz",
 MODULEVENDORSTRING,
 MODINFOSTRING1,
 MODINFOSTRING2,
 XF86_VERSION_CURRENT,
 ZZZ_MAJOR_VERSION,
 ZZZ_MINOR_VERSION,
 ZZZ_PATCHLEVEL,
 ABI_CLASS_VIDEODRV,
 ABI_VIDEODRV_VERSION,
 MOD_CLASS_VIDEODRV,
 {0,0,0,0}
};
```

  The MODULEVENDORSTRING would usually be i "The XFree86 Project" or the corresponding information for the vendor creating this module. The two MODINFOSTRINGs are magical values that identify allow to find the VersionInfo in a module (for example for use in a signing tool). Next comes the version of XFree86 that this module was built again and the version of the module itself. Then the ABI class and version as well as the module class. The final four integers are intended to hold the digital signature of the module.

- a `Setup()` function is needed to integrate the module into the XFree86 loader architecture. This function follows the ABI specification for loadable modules and makes the module functions and data available to the server.

```
static pointer
zzzSetup(pointer module, pointer opts,
         int *errmaj, int *errmin)
```

  The `Setup()` function needs to call `xf86AddDriver()` in order to register the module as a driver.

- an `Identify()` function that prints out some identifying message is mandatory.

```
static void
ZZZIdentify(int flags)
{
 xf86PrintChipsets(ZZZ_NAME,
     "driver for ZZZ Tech chipsets",
     ZZZChipsets);
}
```

- a `Probe()` function is needed that locates the hardware that this driver supports (normally some type of graphics device) and registers this driver as driver for the resource. The Probe should be as unintrusive as possible. It must not modify any settings in the hardware and should not touch any hardware except the devices that are supported by this driver.

```
static Bool
ZZZProbe(DriverPtr drv, int flags)
```

  for most current hardware the `Probe()` function can use information from the PCI data to verify if a supported card is available. A helper function `xf86MatchPciInstances()` is available to make matching PCI devices easier.

- a `PreInit()` function collects all the information that is necessary to determine the configuration of the hardware and to prepare the device for being used, without making any changes to the device at this stage. Again this function should be as unintrusive as possible.

```
static Bool
ZZZPreInit(ScrnInfoPtr pScrn, int flags)
```

  Information that is filled in here includes things obtained from the XF86Config file (e.g., the Monitor info, the amount of video memory, if it was given in the config file), defaults for color depth and bits per pixel (bpp), visual, gamma correction, etc.

  Additional, non-intrusive probing of hardware is done for that type of information that has not been given in the config file (e.g., exact type of chipset, amount of video memory, etc.).

  Next the video modes given in the config file are validated against the restrictions that the hardware puts on video modes.

  Finally, this function loads the submodules necessary for driving the hardware (e.g., the vgahw module for VGA compatible cards) and the framebuffer code.

- Save() and Restore() functions provide a means to save and restore the complete video state of the device. While these functions are not mandatory they are a very useful abstraction of a commonly needed task.

```
static void
ZZZSave(ScrnInfoPtr pScrn)

static void
ZZZRestore(ScrnInfoPtr pScrn)
```

- a ModeInit() function is used to setup a new video mode. This is again a recommended function but not part of the mandatory interfaces.

```
static Bool
ZZZModeInit(ScrnInfoPtr pScrn,
            DisplayModePtr mode)
```

- a ScreenInit() function is needed to hook the driver into the server and to setup the initial video mode (using ModeInit()). This is (in the flow of control) the first function that should modify the device.

```
static Bool
ZZZScreenInit(int scrnIndex,
              ScreenPtr pScreen,
              int argc,
              char **argv)
```

  This functions need to initialize the device independent parts of the X server as well. This includes informing setting up the visual type (miSetVisualTypes()), initializing the framebuffer (by calling the framebuffer ScreenInit() function, e.g. cfbScreenInit()), creating the initial color map, etc.

- EnterVT() and LeaveVT() functions are needed to allow switching to and from the X server.

```
static Bool
ZZZEnterVT(int scrnIndex, int flags)

static void
ZZZLeaveVT(int scrnIndex, int flags)
```

- a SaveScreen() function is mandatory to blank the screen.

```
static Bool
ZZZSaveScreen(ScreenPtr pScreen,
              Bool unblank)
```

- a CloseScreen() function is used to reset the device to its original state.

```
static Bool
ZZZCloseScreen(int scrnIndex,
               ScreenPtr pScreen)
```

Once these functons are implemented and hooked together, a non-accelerated driver for the device is available and can be tested. Adding accelleration to this driver (using the XFree86 Acceleration Architecture XAA) is relatively straight forward and again well documented in the XAA documentation which can be found at `xc/programs/Xserver/hw/xfree86/xaa/XAA.HOWTO`

Summary and Conclusion

In summary, on the one hand, the documentation of the new ddx (devive dependent X) design in XFree86 has significantly improved over the previously available material. On the other hand, the structure and layering of the server itself has become much more straight forward. Many of the previously existing SVGA-isms have been removed, many other implicit assumptions about the design of the bus and the existence of one single graphics card are gone as well.

Following the design document it is fairly straight forward to implement a multi head capable driver for almost any type of graphics device.

The DDK

The next major step forward will be the advent of a real DDK, a real driver development kit. As of this writing, this is not finished, since we need to release 4.0 before we can focus on the DDK. But the design of XFree86 4.0 is finished, the initial release will happen before this paper is going to print, and in summer 2000 a release of a full DDK is planned.

So while this paper will have to skip the description of the DDK, I should be able to talk about this during my presentation at the conference.

5 Availability

You can get the XFree86 sources from our web site `http://www.XFree86.Org`.

References

[1] `http://www.x.org`

[2] `http://www.xfree86.org`

[3] Jim Gettys, Philip L. Karlton, Scott McGregor, *The X Window System Version 11*, DEC CRL 90/08.

[4] Elias Israel, Erik Fortune, *The X Window System Server*, Digital Press.

[5] *in the XFree86-3.x sources:* `xc/programs/Xserver/hw/xfree86/VGADriverDoc`

Linux on the System/390

Adam Thornton
Sine Nomine Associates

Abstract

This paper is meant to serve as a general overview of the Linux port to the IBM System/390 mainframe architecture. The System/390 architecture is introduced, and its history and design are briefly discussed, including IBM's operating system VM, which allows virtualization of the System/390 and hence a way to split the machine into a large number of virtual machines. The short history of Linux on the platform is then covered, and ways in which running Linux on the System/390 make sense are discussed, chief among them the possibility of running many Linux instances on a single System/390.

Differences between the System/390 port and other ports of Linux are introduced, and the necessity for a general solution to the problem of timer interrupts in a virtual environment is raised. The I/O model of the System/390 is then described, with an example comparing a Linux/390 network driver with a PCI network driver that highlights some of the idiosyncracies of the System/390. Ways that a developer can begin working with Linux for System/390 are suggested, ranging from use of an employer's existing machine through acquisition of a used development machine to Hercules, a free System/390 emulator for Linux. Finally, areas in which development help is most badly needed are spotlighted.

Introduction

The System/390 is one of the more recent targets for a Linux port. IBM's port became available in mid–December of 1999, and formal support for it was announced May 17. Further commitments have been made in the meantime by IBM. IBM is clearly heavily invested in supporting Linux on all its platforms (with the exception of the AS–400), and that most definitely includes the System/390. The System/390 port looks, from user mode, like any other Linux system; inside the kernel, and especially inside the device model, it's somewhat unusual. Although mainframes are typically very expensive computers, hobbyists and people wishing to evaluate Linux/390 without investing in a System/390 can use Hercules, a free System/390 emulator, to see Linux on the System/390 with no investment but their time.

History and Architecture of the System/390

The System/390 is the direct descendent of IBM's System/360 series, announced in 1964. The binary format of the instructions has not changed, so it should, in theory, be possible to run a program written for a 360/25 in 1965 on a modern Multiprise 3000. In practice it is more difficult, since although the instructions themselves have not changed, the interface to operating system services certainly has as operating systems have come and gone.

In the mainframe world, you will hear about at least two operating systems. OS/390 is what is usually used for heavy–duty commercial work; it's the successor of MVS, which is the successor to OS/360. Most sites doing serious data processing on the System/390 use OS/390. However, more important from the Linux perspective is VM, "Virtual Machine." VM virtualizes the System/390 hardware. The usual shell run under VM is CMS, the Conversational Monitor System. In essence, each user gets his or her own copy of a System/390, with attached peripherals, to play with. If you've used VMWare on Intel, VM is the same idea, but with 30 years longer to mature, and implemented on hardware that is friendly to self–virtualization.

VM is much more conservative of system resources than VMWare. A typical CMS installation on a large system can support between ten and twenty thousand simultaneous users, each with a unique (virtual) System/390. VM, however, doesn't have to run CMS. Because the interface it presents is, simply a System/390 as defined in the *Principles of Operation*, it can be used to run TPF, VSE/ESA, OS/390[1], and, of course, Linux.

The architecture itself is a fairly standard big–endian 32–bit design. You will often hear System/390 operating systems referred to as 31–bit: this refers to the amount of memory immediately addressable. This is 31 rather than 32 bits because the 360 and 370 architectures were 24–bit designs, and to maintain compatibility when XA ("eXtended Addressing") mode came along, the top bit in a word was set high to indicate XA rather than 370 mode. Additionally, the ESA architecture allows for up to 16Tb of expanded storage, which is accessed by a fancy version of bank switching. Linux on the System/390, however, is still restricted to a 31–bit address space for the time being. The largest usable machine appears to be 1919M

[1] This is a very common use of VM: you can test a new version of OS/390 under VM and work out the migration bugs so when you upgrade the production machine, there is very little downtime of the primary system.

(which is 2G minus 128M minus 1M, but I don't know what significance that has).

If you need a larger machine, currently the only way to go beyond 1919M is to use the XPRAM driver to put swap space in expanded memory (which can lie beyond 2G). This limitation will vanish at the end of this year, when IBM introduces its successor to the S/390 line. It is code–named "Freeway", and not much is known about it (at least by me) other than that it is a 64–bit architecture, and it will presumably include compatibility modes to allow older software to continue to run. A 64–bit Linux port is expected almost immediately on its introduction. Therefore, don't expect to see much action from Boebligen on fixing the 1919M limitation; the recommended solution, if you need that much real memory, will probably be to buy a 64–bit engine.

You've probably heard of EBCDIC. It is the character encoding used by traditional mainframe operating systems. Its primary feature is that it is not ASCII; interoperability with ASCII systems therefore requires translation tables and is a bit of a headache. Linux/390, however, is not (unlike Unix System Services, a platform to make porting Unix applications to OS/390 and VM easier) EBCDIC; it's a plain old ASCII system. As far as the hardware is concerned, it's all just ones and zeroes. The only place you see the EBCDIC translation is in the console driver, as the console expects to see EBCDIC characters appear on it.

The System/390 has two classes of instruction: problem mode and supervisor mode. These correspond to user and kernel states, and allow for privileged instructions which cause a machine trap when a user–mode program tries to execute them. The instruction set is very, very rich: this is an extremely CISCy machine (the current 390 Principles of Operations, or architecture definition, is the size of a telephone book). For this reason, and because IBM provides a very good macro assembler, a great deal of programming on the System/390 is still done in assembly language rather than a high–level language.

History of Linux on the System/390

Linux on the System/390 is an idea that has been being kicked around since Linux's earliest days, but not much was done until 1998 or so. Linas Vepstas and others began a port of Linux, called "Bigfoot", which was an implementation that ran on System/370 (the 390's predecessor) and later processors. By early December 1999, Bigfoot would boot and usually load /bin/sh before panicking and crashing.

In mid–December, IBM Boebligen released its port of Linux to the System/390. The IBM port has significant differences from the Bigfoot port. Most notable is that it actually runs. However, it does require that you have a second–generation (G2) or later CMOS System/390 machine, as it uses certain halfword immediate instructions introduced with the G2 architecture. Peter Schulte–Strack has released a set of "Vintage" patches which allow Linux to run on a G1 machine, which opens the door to a wider variety of processors. However, the IBM port is certainly never going to run on a 370.

While Bigfoot was a vastly interesting project, and was developed as a proper Open Source project rather than as a skunkworks secret endeavor, it appears to be dead, or at least in stasis. All further mention of Linux on the System/390 will assume the IBM port.

The System/390 port was originally bootstrapped by writing a cross–compiling backend to GCC that produced System/390 code, cross–compiling glibc for the 390 architecture, and then uploading those and the kernel built with them to a real System/390. At this point there are two distributions of Linux for the System/390, so it is no longer necessary to build a cross–compilation environment: simply allocate enough disk space, copy Linux/390 onto it, IPL (Initial Program Load, or IBM for "boot"), and go.

Operational Characteristics of the System/390

The System/390 has more I/O capacity than any other computer on the planet. IBM has spent 35 years evolving this line of computers to support enormous databases with rock–solid reliability.

However, the System/390 does not have particularly good CPU horsepower. That's not to say you don't get reasonable performance out of one: they have quite a bit of oomph, but in terms of MIPS per dollar, Intel or Alpha beats them hands–down. They certainly are not cheap, and if what you need is computation rather than I/O, then it makes no sense to run Linux on a System/390. Further, floating–point support on pre–G5 models is not IEEE floating–point. Since Linux expects IEEE floating–point, those instructions must be emulated for earlier processors, which causes a very noticeable speed reduction on anything requiring much floating–point[2], most notably KDE.

However, much of the time, CPU performance is not what is needed, and I/O is. Web hosting is the obvious

[2] Those old enough to remember Linux on the i386 or the i486SX remember the pain of emulated floating–point.

example; in fact, pretty much any sort of e–commerce application is going to have far more significant I/O needs than CPU needs.

One of the features of the System/390 is the ability to split a single physical machine into many virtual machines. This can be done in hardware, with a facility called LPAR ("Logical PARtition"), which allows up to 15 machines to be carved out of a single computer. It can also be done in software. Traditionally, this has been done by running VM.

David Boyes ran a test under VM (VM itself was running in an LPAR of a medium–to–large System/390, from which it could use a maximum of 10% of the machine's cycles) in which he brought up 41,400 simultaneous Linux images before the virtual machine ran out of resources. Although this number is not representative of a real workload (it was Apache serving static pages only), there is a customer running over 3200 Linux machines (as of August 5), in production, on a single System/390.
On August 2, IBM announced VIF, the Virtual Image Facility, which is to all intents and purposes a stripped–down VM, with the ability to run multiple virtual machines, but with the sophisticated monitoring and resource allocation tools removed. VIF would let you run hundreds or thousands of images; however, it would not let you allocate resource caps to particular machines, so it would not be possible to keep one greedy Linux user from affecting performance for the rest of the Linux images on the machine. Under VM it is easy to cap each machine's resource usage, and to change those caps on the fly.

For OS/390–only shops, Linux has been reported to run under ISX, which is essentially a virtual machine facility for OS/390. ISX is not a complete implementation of a System/390: notably, it doesn't do SIE, and therefore cannot run VM, but it implements enough of the architecture to run Linux.

Why would you want to run Linux on the System/390? There are several good reasons. First is that VM (or VIF) gives you the ability to run thousands of virtual machines of the same piece of hardware. For an ISP or ASP, the savings in terms of facilities and management costs quickly overcome the higher initial cost of the System/390 hardware: if you are building a data center based around Suns, your crossover point is around 25 servers; with Intel, it's more like 150 servers. In either case, this is a small fraction of the several thousand machines VM can run with acceptable performance.

Networking many virtual machines is much easier than managing a physical facility. All you need to do is to pick a machine that owns the actual network interface, or define one machine per interface if you prefer. This machine has traditionally been the VM TCP/IP virtual machine (that is, the virtual machine in charge of VM's TCP stack), but there is no reason it couldn't be an OS/390 LPAR or, indeed, a Linux virtual machine. That machine is then set up as the gateway, providing route information via routed about the machines behind it. If you preferred, and if your network was simple enough, you could, of course, use static routes. The virtual machines then run with point–to–point connections over virtual channel–to–channel connections or over VM's Inter–User Communications Vehicle (IUCV). vCTC speeds are about 250MB/s, and IUCV about 500MB/s, sustained.

For certain webhosting models, the ability to bring a new server online for a customer within 90 seconds at a marginal cost of very nearly zero is a compelling argument. For commercial hosting you're probably going to want to run under VM rather than VIF, so that you can guarantee SLAs by preventing resource hogging. On the other hand, if what you want to do is server consolidation without guaranteed resource limits, VIF would be ideal. For instance, you might want to take a bunch of departmental servers and place them on a single box under unified management, while preserving departmental autonomy by giving each department its own virtual machine.

A university (universities are traditionally good places to find underutilized mainframes with pre–existent VM licenses) might want to run an advanced operating systems or networking class where you can give each student his or her own machine. The student can have absolute control over that machine, and thus you can do interesting systems programming assignments without the risk of a traditional shared machine setup and without the cost of giving each student a physical machine. The University of Nebraska is, in fact, doing exactly this. Academic clustering research can also benefit from Linux under VM, since it's a lot easier and cheaper to bring up multiple images on the same physical box (assuming the mainframe is already installed at the site) than it is to wire a bunch of machines together.

Finally, it even sometimes makes sense to run only a single Linux image on the System/390. This might be the case if your shop runs a database on OS/390 (or VM) already. By putting up Linux, either in an LPAR or under VM, you gain all the benefits of running Apache as your Web front end: it's free, it's easy to find staff with administrative skills, you can use Perl, Python, PHP, or even ASP as a scripting environment, and, best of all, the network connection between your Web server and your database runs at memory speed over vCTC or IUCV.

Any multi–tier application that depends on a System/390 as its back end benefits from this approach. This does not have to be a web server/database combination. For example, if you had a POP or IMAP server residing on your mainframe providing mail services to users, you might want to implement spam and relaying protection in your SMTP MTA. While you could code this from scratch, sendmail already knows how to do it: it might be much more efficient to run sendmail in a virtual machine to provide this protection and then hand the remaining mail off to your native mail server.

The only reason I can think of to run Linux native on a real System/390 as the only operating system would be if you have a very low–end System/390 as a development machine (assuming the P/390 console problem has been solved), or if you have a System/390 which has been decommissioned, and thus you no longer have OS/390 or VM for it, but want it to do something useful until it develops a hardware problem. Of course, if you're using Hercules, and thus getting an emulated System/390 for free, Linux makes a lot of sense; this is really a special case of a low–end development box.

The Kernel

The kernel is pretty much under the control of IBM Boebligen. However, it also largely works, so there really isn't very much to do.

There are two outstanding issues, solutions to both of which are rumored to be under development by IBM.

The first, and most significant, concerns the timer interrupt. The default value of HZ on the System/390, as on all other architectures except the Alpha, is 100[3]. In normal operation, an interrupt handler executed 100 times a second is not a problem. However, consider the case where you have 5000 virtual machines all executing at once. Then you're needing to service half a million interrupts each second, and that eats heavily into the amount of processor time available to get useful work done. In fact, both for the 41,400 image test and for the 3200 virtual machines in production, the value of HZ has been set to 10. This means that interactive performance is atrocious, although since the production servers are running only INN and bind, their performance is still adequate for their workload.

[3] I've heard that HZ changes to 1000 in 2.4, at least for the Intel processor. This would be a very bad idea for the System/390, especially if it is running a virtual environment, as you will see.

The right answer to this problem is to have something like `#ifdef RUNNING_VIRTUAL` in the kernel, which disables timer interrupts in the idle task, and upon receiving a real interrupt, sets the jiffy count by querying its hypervisor to ask for the real time of day. This would not just be a VM fix: it is equally applicable to Linux under VMWare, under Plex86, and to User–Mode Linux.

The second issue involves PAGEX support. Currently, when Linux pages, the current Linux task is suspended while the right page is brought into memory. PAGEX comes from the VM world, and would allow the Linux machine to signal to its hypervisor that the machine is page–faulting. That would not only enable us to do away with one level of paging (VM's paging is much more sophisticated than Linux's), but would enable VM to more efficiently schedule its tasks, which might include multiple Linux machines.

System/390 support was integrated into the main kernel tree as of 2.2.14. Whether it works right out of the box is another matter. 2.4 support should be along Real Soon Now. The kernel development keeps tickling hitherto undiscovered bugs in GCC and glibc, and high optimization levels and thread support are still a little buggy. The assembly syntax used owes a lot more to GCC than to the mainframe, so old–time mainframers may feel a bit out of place, but anyone who has worked with assembly code under GCC will feel right at home, once the new instruction set is digested.

The only truly odd part of the System/390 architecture to the developer steeped in PC culture is the I/O subsystem.

System/390 I/O

System/390 devices are attached to channels, which are essentially device busses. Traditionally channels were parallel bus–and–tag architectures, supplied on huge cables[4], but this has been supplanted in the past decade by fiber optic ESCON channels. In any event, they've always been very fast compared to their contemporaries. One way to think of System/390 peripherals is as SCSI on steroids. For an I/O

[4] The term given to reconfiguring physical devices by rewiring them to different channels, typically underneath the raised floor of a machine room, was "poking the boa" (as in, "I can't go have a beer with you Friday; I have to poke the boa."). One of the leading channel cable manufacturers called itself "Anaconda." The names are pretty accurate, in terms of the cable size.

operation, the host processor issues a command to the device (possibly a single command, but more likely an entire channel program), and the device proceeds with its work asynchronously. Then once the device has completed the operation, it will send an interrupt to the processor stating that it's finished its task. Channel commands are documented in *Principles of Operation* as well. Each device has its own subchannel, and up to 65,536 devices are attachable simultaneously.

This makes writing device drivers for Linux/390 more akin to writing SCSI drivers than to writing any other sort of device driver. It's certainly a black and esoteric art, which requires intimate knowledge not only of the Linux device driver structure (Allesandro Rubini's book *Linux Device Drivers* is very helpful here), but of S/390 channel architecture and device design.

A description of the abstraction of the System/390 device model and the API that device driver authors should write to is found in *kernel–source*/Documentation/s390/cds.txt, and is required reading if you're going to be working on device drivers. To summarize: the IRQ does not exist as such on the System/390. However, to modify as little of the extant (mostly x86–based) code as possible, the developers decided to map System/390 subchannels onto IRQs. Basically, instead of being restricted to 16 IRQs, as you are with the ISA bus and its derivatives, under Linux/390, you get 65,536 IRQs to play with.

However, bolting the System/390 I/O model onto Intel assumptions is not without peril. Perhaps the most confusing piece to people who write Intel device drivers is that the dev_id parameter has been reused. While on the Intel architecture, it is used to specify multiple devices sharing an interrupt, on System/390 dev_id serves as a shared buffer to let the generic interrupt layer and the device–specific driver communicate about the status of the interrupt. There are some other differences that can easily bite the developer; because disabling an interrupt on the System/390 actually means telling the device "do not accept any more interrupts" rather than masking a bit on the PIC, disable_irq() is problematic, since we would like it to be able to return an error condition. Thus it and enable_irq() are defined to return int rather than void, which is Intel's behavior.

All actual work done in the device driver is done through the do_IO() interface; the device driver may not directly issue commands on its own. Aside from that, writing a device driver is pretty straightforward: disable the interrupt you're currently handling, schedule a bottom half if it's going to be a long–running process, do your work, reenable interrupts. Of course, you should try to avoid setting the DOIO_WAIT_FOR_INTERRUPT flag, which turns on synchronous processing, since that will cause other CPUs to spin in an SMP environment, and few production System/390s are uniprocessor[5].

Take, as an example, the difference between how bits are put on the wire with *kernel–source*/drivers/net/ne2k–pci.c and *kernel–source*/drivers/s390/net/ctc.c. The PCI driver uses a function called ne2k_pci_block_output (Listing 1), which messes with a bunch of setup (including some little–endian ugliness), sets ei_status.dmaing to indicate the DMA channel is busy, then writes a bunch of 16–or–32 bit chunks, depending on whether it's in 16 or 32–bit mode. It does this with the outsl or outsw functions. Finally, it resets ei_status.dmaing and exits.

On the other hand, the corresponding function in ctc.c, which describes I/O through a 3088 channel interface (which is what is emulated in a vCTC), is called ctc_tx (Listing 2). It too does a lot of setup, mostly related to whether or not the device is currently busy. If it isn't, the spinlock is set to assure synchronization across multiple CPUs, and device transmittal status is set to busy with ctc_test_and_setbit_busy(). The packet to be transmitted is moved into the lp field of the driver's privptr data structure (of type ctc_priv, basically a representation of the 3088's state), and a whole pile of status is set via the channel[] data structure to represent the I/O occuring on this device. The actual write is anticlimactic: do_IO() is called, with privptr supplying all the commands and data, which has the effect of calling WRITE with the contents of lp being put on the wire. Then the busy flag is turned off, the spinlock is unset, and the function returns.

In short, writing device drivers is kind of a mess. Fortunately, there are a few examples in the kernel source tree under *kernel–source*/drivers/s390, and some support functions can be found in *kernel–source*/arch/s390. Be warned that many of the Boebligen developers do not believe in comments; it's not easy going. Once you have device specifications, so you know what channel commands to issue to achieve the desired results, it's not particularly difficult, or at least, no more so than writing device drivers for any other architecture usually is.

Some devices have their channel interfaces well–documented. In that case, writing the device driver is a matter of implementation. It may be a little

[5] Indeed, the uniprocessor support in Linux/390 is still somewhat buggy; it is recommended to build an SMP kernel even for uniprocessor systems because of some unwanted interaction with the network drivers on non–SMP systems.

technically difficult, but there's nothing especially groundbreaking about it. For devices with proprietary interfaces, you're potentially in trouble. If you really feel a need to write a device driver for an undocumented interface, then you'd better hope you're running under VM. If you are, then VM's debugging facilities will let you trace and record input and output to the device, and you can begin to reverse–engineer the communication protocol that way. One good example might be the Shared File System (SFS) for VM. IBM published a set of Rexx scripts that allow you to do SFS over TCP/IP. Armed with that knowledge and a packet sniffer, it would not be very hard to determine what the proprietary SFS communication protocol really consists of. Of course, you can always mount SFS filesystems via the VM NFS server, so the utility of the time investment would be questionable at best.

User Mode

There's really not much at all to say here. It's Linux. There's no EBCDIC in sight. It looks, tastes, feels, and smells like Linux. Porting user–mode applications is generally trivial; if the application does not depend on hardware that isn't available on the System/390, and if it doesn't make any assumptions about byte order, it should, and almost always does, just compile and work. For now it's necessary to patch config.sub and config.guess to recognize Linux/390; autoconf will eventually include system definitions for it, and it's a two–line patch to each file in any case.

How Can I Play?

There are several approaches to developing for Linux/390. Some require access to a real System/390, and some don't.

The easiest, of course, is if your organization already has a System/390. If you're running VM, then it's probably not going to be a problem to allocate a 64M class G (general user) machine with a few hundred megabytes of disk. If not, then it might be a little tougher to get an LPAR, but if you request it and have it happen at your next scheduled maintenance window, it shouldn't be impossible.

If you work at a well–funded shop serious about getting into the Linux/390 market, then you really should invest in a VM license. The ability to create multiple images as well as the debugging and monitoring facilities make development a great deal easier.

If you don't have a System/390, then you might want to acquire one. The smallest machine, currently, is the Multiprise 3000. Even through the IBM Partners in Development program, it's going to cost you $65,000 or more. You may have more luck on the used market. A PCI–based P/390, which is a System/390 on a full–sized PCI card, cost me $5000. MCA versions are cheaper (as little as $800–$1000), but require an MCA PS/2 or RS/6000 to run, which I didn't have. Not all hosts are suitable for a P/390, but the IBM PC Server 325 was popular on Onsale.com last year, and works just fine (the 330 and the 500 were the officially supported platforms). Expect to pay $700 or so for one. Then you need an OS/2 license for the host machine (the P/390 is hosted by OS/2 on the PC, AIX for the RS/6000); however, Warp Server in the shrinkwrap is available for next to nothing from the usual auction sites. In essence, you're looking at about $6000 for the hardware and necessary glue code. Make sure that your P/390 comes with the Licensed Internal Code that supplies the System/390 microcode, or what you have is a very expensive, albeit pretty, paperweight, not a computer.

There's also a commercial software solution: Fundamental Software publishes Flex–ES, which is a System/390 emulator for Intel. Their prices start at about $15,000, but on a high–end Intel system, you can get quite a bit more CPU power than you can from a P/390.

By the time you read this, Linux/390 will almost certainly be booting natively on the P/390. As of the time of writing (early August) there were still some minor problems with the console driver that required setting a hardware breakpoint and manually clearing the registers to get Linux to boot without VM on the P/390.

A VM license will set you back many thousands of dollars. If you're doing this on the cheap, don't go there. However, there's one more solution, which is very slow, but has the great advantage of being free.

Roger Bowler wrote (and Jay Maynard now maintains) Hercules, a System/370 and System/390 emulator for Linux. It emulates the hardware well enough to boot Linux/390 (and, unlike ISX, well enough to run VM), and there exists a device driver (essentially a dummy network driver) for the host system that can make the emulated System/390 think it has a 3088 CTC connection to the host, so you can run TCP/IP applications to talk to the host (e.g. to install SuSE via NFS). The Hercules license allows non–commercial use for no charge.

Hercules is quite slow compared to the alternatives, but on a fast PC it has begun to approach the speed of a first–generation P/390. And, of course, it doesn't cost anything. It's certainly the best evaluation platform available, although it may not be sufficient for serious development work. At least you can get a feeling for what Linux on the System/390 looks like and whether it's something you're interested in pursuing.

You could also build a cross–compilation environment and port software that way. But that's no fun, it's impossible to test in the absence of a System/390, and with Hercules available, there's really no reason to do so.

One other thing to try, if you're serious about Linux/390 work: call IBM. They made me a very attractive deal on a development system, and threw in a VM license. They badly want people developing for the platform, and are certainly serious about making life easier for their developers.

Installing Linux/390

Once you have a G2–or–later System/390, either real or emulated, with a few hundred megabytes of disk space, then you need to choose a distribution. At the moment your choices are Marist College's sort–of–Red–Hat based Linux or SuSE. TurboLinux is on the way but is not yet available.

Basically, this is like installing any other Linux. You prepare the disk space you're going to use (if you're using CMS minidisks, you need to format and reserve them within CMS before booting Linux), boot either from tape or, under VM, the virtual card reader, set up your network devices so you can get to installation media on other machines, use mkfs to build new filesystems (currently ext2 only, although SuSE may contain support for reiserfs by the time you read this), and put the files onto those filesystems. In the case of the Marist distribution, this means unpacking a giant tarball and then editing the configuration files in place; for SuSE, it's just like every other SuSE installation: mount the CD–image somewhere it's accessible via ftp or NFS, select the packages you want, and install.

What Can I Do?

Most of the effort right now needs to focus on device drivers. There are two glaring needs at the moment.

First, Linux desperately needs an open–source network device driver. Because IBM still realizes substantial profits from licensing its OSA–2 network interface design to other companies, the ethernet driver is object code only[6]. Work is being done on an Open–Source device driver to speak the CLAW protocol, which would enable Linux to talk to a channel–attached router (e.g., any Cisco 7xxx router). Hercules can use the ctc driver to talk to a virtual network interface on the host. However, if you want to make a difference fast, write a very dumb, but open–sourced, 3172 network driver for Linux/390. This would require reverse–engineering the 3172 communication protocols and implementing a large enough subset to let you put bytes on and take bytes off the network.

The second need is presumably being addressed by IBM. Although Linux/390 can boot from a tape, it cannot use tape devices. Until there is native tape support, it is difficult to implement a backup solution for Linux/390, although another avenue of approach might be to write or port an Amanda client for VM or OS/390. However, as with any device driver development, familiarity with both Linux and with the hardware platform is necessary.

There are a number of nonessential "wouldn't it be nice ifs." Channel–attached printer support is one such. However, it's not nearly as necessary as a tape driver: lpr works fine, and lpd implementations exist for VM and OS/390.

Although recent versions of mainframe operating systems have finally incorporated TCP/IP, much of the mainframe world still relies on SNA (Systems Network Architecture) to do its networking. There is a Linux–SNA project, and although it is not yet incorporated into Linux/390, it is sponsored by TurboLinux. Since TurboLinux has announced its intention to produce a Linux/390 distribution, it would be surprising if Linux–SNA weren't included. If you know both SNA and Linux, and would like to see Linux (whether on the System/390 or on Intel) penetrate further into traditional mainframe shops, this would be an excellent area to explore.

If you intend to run Linux under VM, improved access to VM facilities (particularly debugging and performance monitoring), would be wonderful. Although Neale Ferguson has written hcp, which allows you to issue CP commands from Linux/390, there's a lot of work still to be done. This too, obviously requires deep acquaintance with the System/390 architecture and with VM's services. If you're trying this route, a CMS minidisk filesystem driver would be a good place to start (even though you can access the data over NFS already).

The System/390 is one of the most reliable pieces of hardware on the planet, with MTBFs of 60 years for recent 9672s. However, Linux high–availability

[6] This is legal because the driver is loaded as a module and not actually linked into the kernel.

software support lags pretty far behind that of Solaris. Red Hat has announced Piranha, an Open–Source software HA project. I'm working on bringing it to Linux/390, and any work you can do on it will help out not just Linux/390 but Linux as a whole.

If you want to work on really bleeding–edge High Availability stuff, consider the "VM Stun" project of David Boyes and Perry Ruiter. VM already supports clustering for multiple boxes in the same location cabled together. Global clustering would stretch that cable. The essence of the idea is that a virtual machine state, and its meta–information (size of the machine, privilege class, attached devices, and so on) would be frozen, the entire package would be shipped over the wire to some other location, and then the package would be thawed and paged into the address space of the new host machine. Saving the virtual machine state is easy, but saving, shipping, and restoring the meta–information is much more difficult.

We're discovering new GCC and glibc bugs almost daily. If you're more adventurous than I am, you could work on these. And, of course, if there's any particular piece of software you want, that doesn't yet exist for Linux/390, then you're free to port it yourself. Most applications require nothing more than a patch to config.sub and config.guess and then a recompilation. Doom took me about 20 minutes to port. Quake's a little harder, and because it relies on OpenGL (System/390s do *not* have accelerated video), is probably going to be unacceptably slow, even on a fast processor. Before you expend the energy, check out SuSE and the Iron Penguin project to see whether someone else has already ported your application. The odds are good that it's already been done.

If it hasn't, and if you aren't using autoconf to generate an appropriate system definition, it's been my experience that the Linux PowerPC port of anything is usually the right place to start. It's 32 bits, it's big–endian, and it usually doesn't use any inline assembly. Because it's Linux, we can assume GCC and GNU make and the usual GNU tool layout. Once you can specify these parameters in the Makefile, the battle is just about won. High levels of optimization are still pretty buggy, so turning optimization down to –O1 is usually the right thing to do. If even that generates code that doesn't quite work, –O0 may be required. These problems should go away as the bugs get shaken out of the compiler and libraries.

Conclusion

Linux/390 is an exciting and fun platform to develop for. From a user–mode perspective it's not very exciting, since everything behaves as it should and there's little surprise. However, writing device drivers is challenging, and writing tools to interact with a traditional System/390 environment, be that VM, OS/390, or VSE, is difficult and engrossing. The System/390 for the first time puts Linux on machines with the potential for extremely high reliability and enormous I/O throughput. VM (or VIF) gives the ability to run thousands of Linux images on a single piece of hardware, which not only is a cost–effective solution for the ASP/ISP market, but is also a great opportunity to do clustering research much more simply than with a traditional, discrete–machine setup.

Resources

http://linux390.marist.edu
http://www.s390.ibm.com/linux
http://www.vm.ibm.com/linux
http://www.linux390.com
http://linux.s390.org
http://penguinvm.princeton.edu

Listings

Listing 1: ne2k_pci.c, output of bytes to network in ne2k_pci_block_output()

```
/* Now the normal output. */
        outb(count & 0xff, nic_base + EN0_RCNTLO);
        outb(count >> 8,   nic_base + EN0_RCNTHI);
        outb(0x00, nic_base + EN0_RSARLO);
        outb(start_page, nic_base + EN0_RSARHI);
        outb(E8390_RWRITE+E8390_START, nic_base + NE_CMD);
        if (ei_status.ne2k_flags & ONLY_16BIT_IO) {
                outsw(NE_BASE + NE_DATAPORT, buf, count>>1);
        } else {
                outsl(NE_BASE + NE_DATAPORT, buf, count>>2);
                if (count & 3) {
                        buf += count & ~3;
                        if (count & 2)
                                outw(cpu_to_le16(*((u16*)buf)++), NE_BASE + \
                                      NE_DATAPORT);
                }
        }

        dma_start = jiffies;
```

Listing 2: ctc.c, moving of packet into privptr and output of bytes to network in ctc_tx()

```
        (__u8 *)lp = (__u8 *) &privptr->channel[WRITE].free_anchor->block->length \
                              + privptr->channel[WRITE].free_anchor->block->length;
        privptr->channel[WRITE].free_anchor->block->length += \
                  skb->len + PACKET_HEADER_LENGTH;
        lp->length = skb->len + PACKET_HEADER_LENGTH;
        lp->type = 0x0800;
        lp->unused = 0;
        memcpy(&lp->data, skb->data, skb->len);
        (__u8 *) lp += lp->length;
        lp->length = 0;
        dev_kfree_skb(skb);
        privptr->channel[WRITE].free_anchor->packets++;
        if (test_and_set_bit(0, (void *)&privptr->channel[WRITE].IO_active) == 0) {
              ctc_buffer_swap(&privptr->channel[WRITE].free_anchor, \
                  &privptr->channel[WRITE].proc_anchor);
                privptr->channel[WRITE].ccw[1].count = \
                  privptr->channel[WRITE].proc_anchor->block->length;
                privptr->channel[WRITE].ccw[1].cda   = \
                  (char *)virt_to_phys(privptr->channel[WRITE].proc_anchor->block);
                parm = (__u32) &privptr->channel[WRITE];
                rc2 = do_IO (privptr->channel[WRITE].irq, \
                  &privptr->channel[WRITE].ccw[0], parm, 0xff, flags );
                if (rc2 != 0)
                        ccw_check_return_code(dev, rc2);
                dev->trans_start = jiffies;
        }
```

A user-mode port of the Linux kernel

Jeff Dike

Abstract

The Linux kernel has been ported so that it runs on itself, in a set of Linux processes. The result is a user space virtual machine using simulated hardware constructed from services provided by the host kernel. A Linux virtual machine is capable of running nearly all of the applications and services available on the host architecture. This paper describes the capabilities of a user-mode virtual machine, the design and implementation of the port, some of the applications that it lends itself to, and some of the work that remains to be done.

1 Overview

1.1 Description of functionality

The user-mode kernel is a port of the Linux kernel to the Linux system call interface rather than to a hardware interface. The code that implements this is under the arch interface. So, this kernel is a full Linux kernel, lacking only hardware-specific code such as drivers.

It runs the same user space as the native kernel. Processes run natively until they need to enter the kernel. There is no emulation of user space code.

Processes running inside it see a self-contained environment. They have no access to any host resources other than those explicitly provided to the virtual machine.

1.2 Device support

All devices seen by the user-mode kernel are virtual from the point of view of the host. They are constructed from software abstractions provided by the host. The following types of devices are provided:

Consoles The main console is whatever terminal the kernel was invoked in. In addition, virtual consoles are supported. By default, they execute an xterm when opened. Optionally, they can just allocate a pseudo-terminal which the user can connect to with a terminal program such as minicom or kermit.

Block devices The block device driver operates within a file on the host. Normally, this is a file containing a filesystem or swap space. However, any file on the host that is seekable is suitable. So, devices on the host can be accessed through their device files.

Serial lines The serial line driver allocates a pseudo-terminal. Users wanting to connect to the virtual machine via a serial line can do so by connecting to the appropriate pseudo-terminal with a terminal program.

Network devices There are two network device drivers. The old network driver communicates with the host networking system through a slip device that it creates in the host. The virtual machine's side of the connection is a pseudo-terminal in the host which appears as a network device inside. There is also a newer network driver which uses an external daemon to pass Ethernet frames between virtual machines. This daemon can also attach this virtual network to the host's physical Ethernet by way of an ethertap device. With an appropriate packet forwarding policy in the daemon, the virtual Ethernet can be transparently merged with the physical Ethernet, totally isolated from it, or anything in between.

2 Design and implementation

2.1 Overview

The final and most important piece of hardware that needs to be implemented virtually is the processor itself, including memory management, process management, and fault support. The kernel's arch interface is dedicated to this purpose, and essentially all of this port's code, except for the drivers, is under that interface.

A basic design decision is that this port will directly run the host's unmodified user space. If processes are going to run exactly the same way in a virtual machine as in the host, then their system calls need to be intercepted and executed in the virtual kernel. This is because those processes are going to trap directly into the host kernel, rather than the user-mode kernel, whenever they do a system call. So, the user-mode kernel needs a way of converting a switch to real kernel mode into a switch to virtual kernel mode. Without it, there is no way to virtualize system calls, and no way to run this kernel.

This is implemented with the Linux ptrace system call tracing facility. A special thread is used to ptrace all of the other threads. This thread is notified when a thread is entering or leaving a system call, and has the ability to arbitrarily modify the system call and its return value. This capability is used to read out the system call and its arguments, annull the system call, and divert the process into the user space kernel code to execute it.

The other mechanism for a process to enter the kernel is through a trap. On physical machines, these are caused by some piece of hardware like the clock, a device, or the memory management hardware forcing the CPU into the appropriate trap handler in the kernel. This port implements traps with Linux signals. The clock interrupt is implemented with the `SIGALRM` and `SIGVTALRM` timers, I/O device interrupts with `SIGIO`, and memory faults with `SIGSEGV`. The kernel declares its own handlers for these signals. These handlers must run in kernel mode, which means that they must run on a kernel stack and with system call interception off. The first is done by registering the handler to run on an alternate stack, the process kernel stack, rather than the process stack. The second is accomplished by the handler requesting that the tracing thread turn off system call tracing until it is ready to re-enter user mode.

When a process is enters kernel mode, it is branching into a different part of its address space. On the host, processes automatically switches address spaces when they enter the kernel. The user-mode port has no such ability. So, the process and kernel coexist within the same address space. The design of the VM system is partly a question of address space allocation. Conflicts with process memory are avoided by placing the kernel text and data in areas that processes are not likely to use. The kernel image itself is linked so that it loads at 0x10000000. The kernel expects the machine to have physical memory and kernel virtual memory areas. The physical memory area consists of a file mapped into each address space starting at 0x50000000. The kernel virtual memory area is immediately after the end of the physical memory area. Virtual memory, both kernel and process, is implemented by remapping pages from the physical memory file into the appropriate place in the address space.

Each process within a virtual machine gets its own process in the host kernel. Even threads sharing an address space in the user-mode kernel will get different address spaces in the host.

Even though each process gets its own address spaces, they must all share the kernel data. Unless something is done to prevent it, every process will get a separate, copy of the kernel data. So, what is done is that the data segment of the kernel is copied into a file, unmapped, and that file is mapped shared in its place. This converts a copy-on-write segment of the address space into a shared segment.

To balance that awkwardness, the separate address space design allows context switches to be largely implemented by host context switches, with preemption driven by the `SIGVTALRM` timer.

`SIGIO` is used to deliver the other asynchronous events that the kernel must handle, namely device interrupts. The console driver, network drivers, serial line driver, and block device driver use the Linux asynchronous I/O mechanism to notify the kernel of available data.

2.2 Virtual machine initialization and shutdown

Before the kernel itself starts booting, some initialization needs to be done in order to make the process look enough like a real machine that the kernel can boot it up. This is analogous to the boot loader on a physical machine doing some hardware setup before running the kernel.

The process arguments are concatenated into the buffer in which the kernel expects to find its command line. Some arguments, which affect the configuration of the virtual machine and how it's to be debugged, are parsed at this point. The physical memory area is set up, some initialization of the task structure and stack of the idle thread is done, the idle thread is started, and the initial thread settles down to its permanent job as the tracing thread.

The idle thread calls `start_kernel` and the virtual machine boots itself up. There is some more architecture-specific initialization that needs to be done. `mem_init` makes memory available to the initial boot process, `paging_init` makes all free memory available to `kmalloc`, and the various drivers are registered and initialized.

At the other end of the virtual machine's lifespan, when `halt` or `reboot` are run, an architecture-specific routine is eventually called to do the actual machine halt or restart. In this port, that involves killing all processes which are still alive, including any helper threads which weren't associated with any virtual machine processes, and asking the tracing thread to finish the shutdown.

If the machine is being halted, the tracing thread simply exits. If it's being rebooted, it loops back to calling the machine initialization code. At that point, the machine boots back up just as it did when it was first run.

2.3 Process creation and destruction

When a new process is created, the generic kernel creates the task structure for the new process and calls the arch layer to do the machine-dependent part. This port creates a new process in the host for each new process in the virtual machine. This is done by the tracing thread. It needs to ptrace all new processes, and that is simplified if it's their parent.

The new thread starts life in a trampoline which does some initialization. It sets up its signal handlers for `SIGSEGV`, `SIGIO`, and `SIGVTALRM`, initializes the timer, and sets itself to be ptraced by its parent.

Once this initialization is done, it sends itself a `SIGSTOP`. When the tracing thread sees the thread stop itself, it sets the thread's system call return value to zero, while the same system call in the forking process returns the pid of the new process.

At this point, the parent's `fork` or `clone` is finished, and it returns.

At some later point, the new process will be scheduled to run. It is necessary that a process call `schedule_tail` before it's rescheduled. Also, the kernel stack state needed to start a system needs to be saved. Both of these are done at this point. Another signal is delivered to the new process, and the handler calls `schedule_tail`, goes into the system call handler, and stops itself. The tracing thread captures the process state at this point. Then the registers of the forking process (except for the one reserved for the system call return value, which is now zero) are restored, and the process is continued. Since the generic kernel arranged for the new address space to be a copy of the parent address s-pace, and the new process has the same registers as the old one, except for the zero return value from the system call, it is a copy of its parent. At this point, its initialization is finished, and it's just like any other process in the virtual machine.

The other end of a process lifespan is fairly simple. The only resources that need to be cleaned up are some kmalloced buffers in the thread structure, which are freed, and the process in the host, which is killed.

2.4 System calls

System call virtualization is implemented by the tracing thread intercepting and redirecting process system calls into the virtual kernel. It reads out the system call and its arguments, then annuls it in the host kernel by changing it into `getpid`. In order to execute the system call in user space, the process is made to execute the system call switch on its kernel stack. This can be (and has been) done in two different ways.

The first is to use ptrace to impose a new set of register values and stack context on the process. This context represents an execution context which puts the process at the beginning of the system call switch. This context is constructed by the process, just after its creation, calling the switch procedure and sending itself a SIGSTOP. The tracing thread sees the SIGSTOP and saves the process register and stack state in the task structure. When this state is restored and the process continued, it emerges from the call to `kill` that it used to stop itself.

The second is to deliver a signal to the process before it's continued into the `getpid`. The Linux system call path checks for signals just before returning to user space, so that signal will be delivered immediately. The signal handler is the system call switch and it is installed so that it executes on an alternate stack (the process kernel stack). This has the same effect as manually restoring the context, but the host kernel does most of the work.

Regardless of which mechanism is used to impose the kernel execution context on the process, the tracing thread continues it with system call tracing turned off. The process reads the system call and its arguments from its thread structure and calls the actual system call procedure.

When it finishes, the return value is saved in the thread structure, and the process notifies the tracing thread that it needs to go back to user space. The tracing thread stores the return value into the appropriate register and continues the process again, with system call tracing turned back on. Now, the process starts executing process code again with the system call return value in the right place, and the user space code can't tell that anything unusual happened. Everything is exactly the same as if the system call had executed in the host kernel.

2.5 Context switching

If a process sleeps instead of returning immediately from a system call, it calls `schedule`. The scheduler selects a new process to run and calls the architecture-specific context switching code to actually perform the switch. In this port, that involves the running process sending a message to the tracing thread that it is being switched out in favor of another process. Since each process in the virtual machine is also a process in the host, the tracing thread performs the switch by stopping the old process and continuing the new one. The new process returns from the context switch that it entered when it last ran and continues whatever it was doing.

Sometimes, after a process is switched back in, its address space will need some updating. If some of its address space had been paged out while it was sleeping, those physical pages, with their new contents, will still be mapped. So, in this situation, the process will need to update its address space by unmapping pages which are no longer valid in its context. These pages are listed in a circular buffer whose address is stored in the process `mm_struct`. When a page is swapped out from a process while it's asleep, its address is appended to this buffer. When the process wakes up, it looks at this buffer to see if there are any changes to its address space that it hasn't applied yet. It then updates its address space and sets an index in its thread structure to point to the end of the buffer. This private index is necessary because many processes might share a virtual address space and an `mm_struct`. In general, their host address spaces will be in different states, depending on how long it's been since they've been updated. So, each process keeps track of how many address changes in this buffer it has seen.

There is a possibility that the buffer might wrap around while a process is asleep and some of the address space changes it needs to make have been lost. To avoid this, the index into the buffer that each process maintains is really an absolute count. The index is obtained by dividing the count by the buffer size and taking the remainder. If the buffer has wrapped, then the process count of address space changes will differ from the actual count by more than the number of slots in the buffer. In this case, the entire address space will be scanned and compared to the process page tables. Any differences will be fixed by remapping or unmapping the page concerned.

2.6 Signal delivery

When a signal has been sent to a process, it is queued in the task structure, and the queue is periodically checked. In this port, the check happens on every kernel mode exit. If a signal needs to be delivered, a SIGUSR2 is sent to the underlying process. The SIGUSR2 handler runs on the process stack, and it executes the process signal handler by simply making a procedure call to it.

Things are a little more complicated if the signal is to be delivered on a different, process-specified, stack. In this case, the alternate stack state, which is composed of register values and stack state, is imposed on the process with ptrace. This new state puts the process in the signal delivery code on the alternate stack, which invokes the process signal handler.

2.7 Memory faults

Linux implements demand loading of process code and data. This is done by trapping memory faults when a nonexistent page is accessed. In this port, a memory fault causes a SIGSEGV to be delivered to the process. The segmentation fault handler does the necessary checking to determine whether it was a kernel-mode or user-mode fault, and in the user-mode case, whether the page is supposed to be present or not. If the page is supposed to be present, then the kernel's page fault mechanism is called to allocate the page and read in its data if necessary. The segmentation fault handler then actually maps in the new page with the correct permissions and returns.

If the fault was not a legitimate page fault, then the machine either panics or sends a SIGSEGV to the process depending on whether it was in kernel mode or user mode.

An exception to this is when a kernel mode fault happens while reading or writing a buffer passed in to a system call by the process. For example, a process may call **read** with a bogus address as the buffer. The fault will happen inside the kernel, in one of the macros which copy data between the kernel and process. The macro has a code path that executes when a fault happens during the copy and returns an appropriate error value. The address of this path is put in the thread structure before the buffer is accessed. The fault handler checks for this address, and if it is there, it puts the fault address in the thread structure, copies the error code address into the ip of the sigcontext structure, and returns. The host will restore the process registers from the sigcontext structure, effectively branching to the error handler. The error handler uses the fault address in some cases to determine the macro return value. When this happens, the system call will usually react by returning EFAULT to the process.

2.8 Locking

There are three types of locking. On a uniprocessor, interrupts must be blocked during critical sections of code. In this port, interrupts are Linux signals, which are blocked and enabled using sigprocmask.

SMP locking involves a processor locking other processors out of a critical section of code until it has left it. The instructions needed to do this are not privileged on i386, so the SMP locking primitives are simply inherited from the i386 port.

The same is true of semaphores. The i386 semaphore primitives work in user space as well as in the kernel, so they are inherited from the i386 port.

2.9 IRQ handling

The IRQ system was copied verbatim from the i386 port. It works with almost no modifications. When a signal handler is invoked, it figures out what IRQ is represented by the signal, and calls do_IRQ with that information. do_IRQ proceeds to call the necessary handlers in the same way as on any other port.

The one difference between this port and others is that some IRQs are associated with file descriptors. When input arrives, the SIGIO handler selects on the descriptors that it knows about in order to decide what IRQs need to be invoked.

3 A virtual machine

The result of all of this infrastructure is that the Linux kernel runs in a set of Linux processes just as it does on physical hardware. The machine-independent portions of the kernel can't tell that anything strange is happening. As far as they are concerned, they are running in a perfectly normal machine.

When user-mode Linux is started, the normal kernel boot-up messages are written to its console, which is the window in which it was run. When the kernel has initialized itself, it runs `init` from the filesystem that it's booting from. What happens from that point is decided by the distribution that was installed in that filesystem.

Essentially all applications that run on the native kernel will run in a virtual machine in the same way. Examples include all of the normal daemons and services, including the Apache web server, `sendmail`, `named`, all of the network services, and X, both displaying as a client on the host X server, and as a local X server.

4 Applications

4.1 Kernel debugging

This port was originally conceived as a kernel debugging tool. A user-mode kernel port makes it possible to do kernel development without needing a separate test machine. It also enables the use of the standard suite of process development and debugging tools, like `gdb`, `gcov`, and `gprof`, on the kernel.

A difficulty with using `gdb` is that the kernel's threads are already being ptraces by the tracing thread. This makes it impossible for `gdb` to attach to them to debug them. Early on, this problem was dealt with by having the tracing thread detach from the thread of interest. Then, `gdb` could attach to it, and it could be debugged as a normal process. Rarely, when this was over, the thread was re-attached by the tracing thread, and the kernel continued. More often, the whole kernel was killed at the end of that debugging session.

Now, there is a mechanism that allows `gdb` to debug a kernel thread without needing to attach to it, and without needing to detach the tracing thread from it. This works by having the tracing thread start `gdb` under system call tracing. The tracing thread intercepts ptrace system calls and a few others made by `gdb`, executes them itself, nullifies the `gdb` system call, and writes the return value into the appropriate register in `gdb`. In this way, `gdb` is faked into believing that it is attached to the thread and is debugging it.

4.2 Isolation

Other uses of this port became apparent later. A number of applications involve isolating users of virtual machines from each other and from the host.

There are several reasons to want isolation. One is to protect the physical machine and its resources from a potentially hostile process. The process would be run in a virtual machine which is given enough resources to run. Those resources would not be valuable, so they could be easily replaced if destroyed. It would be given a copy of an existing filesystem. If it trashes that filesystem, then it would just be deleted, and a new copy made for the next sandbox. It would have no access to valuable information, and its use of the machine's resources would be easily limited. The virtual Ethernet driver also makes it easy to control its access to the net. The daemon that does packet routing could be made to do packet filtering in order to control what traffic the sandboxed process is allowed to send and receive.

A variation on this theme is to put a non-hostile, but untrusted service in a virtual machine. A service is untrusted if it's considered to be vulnerable to being used to break into a machine. `named` is such a service, since it has had at least one hole which led to a spectacular number of breakins. An administrator not wanting to see this happen again would run `named` in a virtual machine and set that machine to be the network's name server. `named` requests from outside would be passed directly from the host to the virtual machine. So, anyone successfully breaking into that service would be breaking into a virtual machine. If they realized that, they'd need to find another exploit to break out of the virtual machine onto the host.

Another use of this isolation is to allocate machine resources, whether they be CPU time, memory, or disk space, between competing users. A virtual machine is given access to a certain amount of machine resources when it's booted, and it will not exceed those resources. So, if a user runs a very memory-intensive process inside a virtual machine, the virtual machine, and not the physical machine, will swap. Performance inside the virtual machine may be bad, but no one else using the physical machine will notice. The same is true of the other types of resources. CPU time can be allocated through the assignment of virtual processors to virtual machines. If a virtual machine is given one processor, it will never have more than one process running on the host, even if it's running a fork bomb. Again, life will be miserable inside the virtual machine, but no one outside will notice.

This level of isolation may find a large market in the hosting industry. Current hosting arrangements vary from application-specific hosting such as Apache virtual hosting to `chroot` environments to dedicated, colocated machines. Dedicated machines are used by customers who want complete control over their environments, but they have the disadvantage that they require a physical machine and they consume rack space and power. Running many virtual machines on a large server offers the advantages of a dedicated machine together with the administrative conveniences of having everything running on a single machine.

4.3 Prototyping

Many sorts of services are more convenient to set up on a virtual machine, or a set of them, before rolling them out on physical machines. Network and clustering services are good examples of this. Setting up a virtual network is far more convenient that setting up a physical one. Once a virtual network is running, new services can be configured and tested on it. When the configuration is debugged, it can be copied to physical machines with confidence that it will work. If it doesn't, then it is likely that the hardware has been set up incorrectly.

So, prototyping first in a virtual environment allows software configuration problems to be separated from hardware configuration problems. If the software has been configured properly in a virtual environment and it doesn't work in a physical environment, there is a high probability that something is wrong with the physical configuration. So, time would not need to be wasted looking at the software; attention would be focussed on the hardware.

Another possibility is to use a virtual machine as a test environment to make sure that a service is working properly before running it on physical machines. This is especially attractive if there are a variety of environments that the new service will need to run in. They can all be tested in virtual machines without needing to dedicate a number of physical machines to testing.

4.4 Multiple environments without multiple machines

It is often convenient to have multiple environments at one's disposal. For example, it may not be obvious that a piece of software will work the same on different Linux distributions. The same is true of different versions of a distribution, library, or service. It might be necessary to test a piece of software in the various environments in which it is expected to run.

Normally, in order to check this, it is necessary to have multiple machines or a single multi-boot machine. However, this port makes it possible to run different environments in different virtual machines, allowing testing in those environments to happen on a single physical machine without rebooting it.

4.5 A Linux environment for other operating systems

When this port is made to run on another operating system, it implements a virtual Linux machine within that OS. As such, it represents an environment in which that OS can run Linux binaries.

A number of operating systems already have some amount of binary compatibility with Linux, and several have Linux compatibility environments. This is potentially another way of achieving the same goal. With Linux becoming the Unix development platform of choice, other Unixes are going to be looking for ways to run Linux executables, and this may be a good way of accomplishing that.

5 Future work

5.1 Protection of kernel memory

Since the kernel's memory is in each process address space, it is vulnerable to being changed by user space code. This is a security hole as well as making the entire virtual machine vulnerable to a badly written process. Kernel memory needs to be write-protected whenever process code is running, and write-enabled when the kernel is running. The one tricky aspect of this is that the code which write-enables kernel data will run on a kernel stack, which needs to be writable already. So, that stack page will be left writable when the process is running. It's not a problem if the process manages to modify it because it is fully initialized before any code starts running on it. Nothing depends on anything left behind on the stack.

5.2 Miscellaneous functionality

At this writing, the user-mode port is nearly fully functional. There's a major omission and some minor ones:

SMP Emulation This is the most serious missing functionality. This will allow a virtual machine to be configured with multiple virtual processors, regardless of how many physical processors are present on the host. This will allow SMP development and testing to happen on uniprocessor machines. It will also enable scaling experiments which test the scalability of the kernel on many more processors than are present on the host. This will possibly allow the kernel to get ahead of the available hardware in terms of processor scalability. SMP emulation is also a way of allocating CPU time to virtual machines. If one virtual machine is given one processor and another is given two, and they're both busy, the two-processor machine will get twice as much CPU time as the uniprocessor machine because it will have two processes running on the host versus the one being run by the uniprocessor machine.

`SA_SIGINFO` support The `SA_SIGINFO` flag allows processes to request extra information about signals that they've received. There is nothing difficult about supporting this option - it just hasn't been needed so far.

Privileged instruction emulation A number of i386 instructions require that the process executing them have special permission from the processor to run them. The most common examples are the `in` and `out` set of instructions and `sti` and `cli`. These make no sense to run as-is because there is no hardware for `in` and `out` to talk to, and because `sti` and `cli` are not how interrupts are enabled and masked. So, they will have to be emulated using the equivalent user-space mechanisms. Very few applications actually use these instructions, which is why this hasn't been implemented yet.

Virtual Ethernet enhancements The virtual Ethernet daemon can exchange packets with the host and directly with the host's Ethernet, but it also needs to be able to talk to its peers on other machines. This will allow a virtual Ethernet to span multiple physical machines without putting the Ethernet frames directly on the Ethernet, allowing a multi-machine virtual Ethernet to remain isolated from the physical Ethernet.

Nesting Currently, the user-mode kernel can not run itself. The inner kernel hangs after getting fifty or sixty system calls into init. To some extent, this is just a stupid kernel trick, but it does have some utility. The main one is testing. It is a demanding process. It exercises many features of the host kernel that aren't used very often. So if it can run itself correctly, that is a good indication that it is functional and correct. The other main use for nesting is testing itself on multiple distributions. There have been bugs and permission problems which show up one some distributions and not others. So, this would be a convenient way of making sure it works on many distributions.

5.3 Performance improvements

5.3.1 Eliminating the tracing thread

The tracing thread is a performance bottleneck in several ways. Every system call performed by a virtual machine involves switching from the process to the tracing thread and back twice, for a total of four context switches. Also, every signal received by a process causes a context switch to the tracing thread and back, even though the tracing thread doesn't care about the vast majority of signals, and just passes them along to the process.

The one thing that the tracing thread is absolutely needed for is intercepting system calls. The current plan for eliminating it involves creating a third system call path in the native kernel which allows processes to intercept their own system calls. This would allow a process to set a flag which requests a signal delivery on each system call that it makes. The signal handler would be invoked with that flag turned off. Once in the handler, the process would examine its own registers to determine the system call and its arguments, and call the appropriate system call function as it does currently. When it returns, it would write the return value into the appropriate field in its sigcontext structure, and return from the signal handler.

At that point, it would return back into user space with the correct system call return value, and, again, there would be no way to tell that anything strange had happened.

5.3.2 Block driver

The block driver is currently only able to have one outstanding I/O request at a time. This greatly hurts I/O intensive applications. Fixing this would involve either having more than one thread to do I/O operations or using an asynchronous I/O mechanism to allow multiple requests to be outstanding without needing to block.

5.3.3 Native kernel

In order to get the best performance with this port, some work will likely be needed in the host kernel. The need for system call interception without context switches has already been mentioned. In addition, access to the host memory context switching mechanism would probably speed up context switches greatly. The ability to construct and modify `mm_struct` objects from user-space and switch an address space between them would eliminate the potential address space scan from context switches.

Another area to look at is the double-caching of disk data. The host kernel and the user-mode kernel both implement buffer caches, which will contain a lot of the same data. This is obviously wasteful, and tuning the host to be the best possible platform will probably require that this be addressed somehow.

5.4 Ports

5.4.1 Linux ports

Currently, this port only runs on Linux/i386. There are Linux/ppc and Linux/ia64 ports underway, but not completed. The other architectures should also be supported. Ports pose no major problems since Linux hides most of the hardware from its processes. The main things that show through are register names. The system call dispatcher needs to extract the system call number and arguments from specific registers, and put the return value in a specific register afterwards. This is obviously machine-dependent. This is handled by using symbolic register names in the generic code and using a architecture-dependent header file to map those names to real machine registers.

The other major portability problem is the system call vector. Most system calls are common to all architectures, although the assignments to system call numbers are different. A minority of system calls are present on some architectures and not others. It is desirable to keep the bulk of the system call vector in generic code while allowing the underlying architecture to add in its own system calls. This is done by having the vector initialization include, via a macro, architecture-specific slots.

There are a few other machine-dependent details like how a `sigcontext_struct` is passed into signal handlers, how a faulting address is made available to the `SIGSEGV` handler, and conversion between `pt_regs` and `sigcontext_struct` structures. These are handled by a small number of architecture-dependent macros and functions.

5.4.2 Other OS ports

User-mode Linux can be ported to other operating systems that have the necessary functionality. The ability to virtualize Linux system calls by intercepting and nullifying them in the host kernel is essential. An OS that can't do that can't run this port. Also needed is the ability to map pages into an address space in multiple places. Without this, virtual memory emulation can't be done. If those two items are available on a particular OS, then this port can probably run on it.

It would be convenient to have the equivalent of `CLONE_FILES`, which allows processes to share a file descriptor table without sharing address spaces. This is used to open a file descriptor in one process and have it be available automatically in all the other processes in the kernel. An example of this is the file descriptors that are used by the block device driver. They are opened when the corresponding block device is opened, usually in the context of a `mount` process. After that, any process that performs operations on that filesystem is going to need to be able to access the file descriptor. Without `CLONE_FILES`, there will need to be some mechanism to keep track of what file descriptors are open in what processes.

Beyond those, this port makes heavy use of standard Unix interfaces. So ports to other Unixes will be significantly easier than ports to non-Unixes. However, those interfaces have equivalents on most other operating systems, so non-Unix ports are possible.

5.5 Access to physical hardware

One potential use of this port in kernel development that hasn't been explored yet is driver writing and debugging. This requires access to physical devices rather than the virtualized, simulated devices that are currently supported. There are two main types of access that would be required:

I/O memory access Linux currently supports, via the `io_remap` facility, mapping I/O memory into a process virtual address space. This would give a driver access to the device's memory and registers.

Interrupts Device interrupts need to be forwarded to the user-mode driver somehow.

The current plan for supporting this involves a stub driver in the native kernel which can probe for the device at boot time. This driver would recognize the device and provide some mechanism for the real user-mode driver to gain access to it, such as an entry in `/proc` or `/dev`. The user-mode driver would open that file and make requests of the stub driver with calls to `ioctl`. The file descriptor would also provide the mechanism to forward interrupts from the stub driver to the user-mode driver. The stub driver's interrupt routine would raise a `SIGIO` on the file descriptor. One potential problem with this would be if the device needed some kind of response to an interrupt from the driver before interrupts are re-enabled. This would preclude that response from coming from the user-mode driver because the process couldn't be run with hardware interrupts disabled. The stub driver would have to respond, which would require that code be put in the host kernel, and it couldn't be tested in user space.

6 Conclusion

The user-mode port of the Linux kernel represents an interesting and potentially significant addition to the kernel pool. It is the first virtual kernel running on Linux, except possibly for VMWare. As such, it places new demands on the host kernel, possibly resulting in new functionality, which may then be used by other applications. This has already happened. Until 2.2.15 and 2.3.22, ptrace on Linux/i386 was not able to modify system call numbers (and on other architectures, it still can't). The demands of this port prompted Linus and Alan to add that feature, at which point several other applications started using it. At least one of those applications was completely impossible beforehand.

Aside from this port, the only options for running virtual machines are VMWare and IBM VM. In both of those cases, potential applications maybe be ruled out for economic reasons or because VM doesn't run on the application platform. The availability of a free virtual machine running on Linux may open up new opportunities that were reserved for mainframes or which just didn't exist before.

The Linux BIOS

Ron Minnich, James Hendricks, Dale Webster
Advanced Computing Lab, Los Alamos National Labs
Los Alamos, New Mexico

August 15, 2000

Abstract

The Linux BIOS replaces the normal BIOS found on PCs, Alphas, and other machines. The BIOS boot and setup is eliminated and replaced by a very simple initialization phase, followed by a gunzip of a Linux kernel. The Linux kernel is then started and from there on the boot proceeds as normal. Current measurements on two mainboards show we can go from a machine power-off state to the "mount root" step in a under a second, depending on the type of hardware in the machine. The actual boot time is difficult to measure accurately at present because it is so small.

As the name implies, the LinuxBIOS is primarily Linux. Linux needs a small number of patches to handle uninitialized hardware: about 10 lines of patches so far. Other than that it is an off-the-shelf 2.3.99-pre5 kernel. The LinuxBIOS startup code is about 500 lines of assembly and 1500 lines of C.

The Linux BIOS can boot other kernels; it can use the LOBOS(ref) or bootimg(ref) tools for this purpose. Because we are using Linux the boot mechanism can be very flexible. We can boot over standardEthernet, or over other interconnects such as Myrinet, Quadrix, or Scaleable Coherent Interface. We can use SSH connections to load the kernel, or use InterMezzo or NFS. Using a real operating system to boot another operating system provides much greater flexibility than using a simple netboot program or BIOS such as PXE.

LinuxBIOS currently boots from power-off to multiuser login on two mainboards, the Intel L440GX+ and the Procomm PSBT1. We are currently working with industrial partners (Dell, Compaq, SiS, and VIA) to port the LinuxBIOS to other machines. According to one vendor, weshould be able to purchase their LinuxBIOS-based mainboards by the end of this year.

1 Introduction

Current PC and Alpha cluster nodes, as delivered, are dependent on a vendor-supplied BIOS for booting. The BIOS in turn relies on inherently unreliable devices – floppy disks and hard disks – to boot the operating system. While there has been some movement toward a net booting standard, the basic design of the netboot as defined by PXE (ref) is inherently flawed, and not usable in a large-scale cluster environment.

Further, while the BIOS is only required to do a few very simple things, it usually does not even do those well. We have found BIOSes that configure all mainboard devices to a single interrupt (from Compaq); BIOSes that do not configure memory-addressable cards to page-aligned addresses(Gateway and others); BIOSes that will not boot without a keyboard attached to the machine (some versions of the Alpha BIOSes); BIOSes that will not boot if the real-time-clock is invalid (Alpha BIOSes). One engineer has reported to us that he had to write his own PCI configuration code because no BIOS extant could handle his machine's bus structure. BIOSes also routinely zero all main memory when they start, which makes finding lockups in operating systems very hard, since on restart the message buffer and the OS image is wiped out.

Another problem with BIOSes is their inability to accomodate non-standard hardware. For example, the Information Sciences Institute (ISI) SLAAC1 reconfigurable computing board will not work with some BIOSes because it does not configure its PCI interface quickly enough. In some cases, on some mainboards, by the time the BIOS is done probing the PCI bus the SLAAC1 card is not even ready to be probed. As a result the BIOS never finds the SLAAC1. Changing the SLAAC1 hardware is not an option, because the speed problem is an artifact of the FPGA that is used for the PCI interface. If we had control over the BIOS, however, we could change it to accomodate experimental boards which do not always work with standard BIOSes.

Finally, BIOS maintenance is a nightmare. All Pentium BIOSes are maintained via DOS programs, and configuration settings for most Pentium BIOSes and all Alpha BIOSes is via keyboard and display (and, in some truly deranged cases, a mouse). It is completely impractical to walk up to 1024 racked PC nodes and boot DOS on each one in turn to change one BIOS setting. Yet this impractical method is the only one supported by vendors.

This situation is completely unacceptable for clusters of 64, 128, or more nodes. The only practical way to maintain a BIOS in a cluster is via the network, under control of an operating system: BIOS configuration and upgrade should be possible from user programs running under an operating system. BIOS parameters should be available via /proc.

2 Related Work

There are a number of efforts ongoing to replace the BIOS. For the most part these efforts are aimed developing an open-source replacement for the BIOS that supports all BIOS functions – in other words, plug-compatible BIOS software. An end goal of most of these efforts is to be able to boot DOS and have it work correctly. Example projects are the OpenBIOS (www.openbios.org) and the FreeBIOS (which can be found at www.sourceforge.net).

Other efforts aim to build a completely open source version of the IEEE Open Boot standard. This work will require the construction of a Forth interpreter; code to support protocol stacks such as TCP/IP; other code to support Disk I/O and file systems on those disks; in short, many pieces that are already available in real operating systems. If the experience of other similar projects are any indication, this BIOS will take almost as much NVRAM as a gzip-ed Linux kernel, while having much less capability. As an example, the Intel BIOS for the L440GX+ mainboard takes 50% more memory (750 KB) than the LinuxBIOS, and is far less capable.

Finally there is one recent effort which is aimed at providing a commercial replacement for the BIOS, called SmartFirmware (http://www.codegen.com/SmartFirmware/index.html). SmartFirmware is also IEEE Open Boot compliant.

While these efforts are interesting they do not address our needs for clustering. The problem with the BIOS is not only that it is closed-source; the problem is that it is performing a function we no longer need (supporting DOS), rather than functions we do need. The limited nature of the BIOS was originally imposed in the days of 8 KBYTE EPROMS; now, given that the BIOS is using most of a 1 MBYTE NVRAM, we ought to be able to do better. The next generation of mainboard NVRAMs will be 2 Mbytes: there is lots of room for a real operating system on the mainboard. Finally, we need to have the option of doing things the BIOS writers can not imagine, such as booting over Scaleable Coherent Interface or managing the BIOS via SSH connections.

3 Structure of the Linux BIOS

The structure of the LinuxBIOS is driven by our desire to avoid writing new code. We want to build the absolute minimum amount of code needed to get Linux up, and then let Linux do the rest of the work. Linux has shown its ability to handle quirky hardware; it is doubtful we can do a better job than it can.

One initial concern for any PC BIOS is what mode to run the processor in. All x86 family processors boot up into an 8086 emulation mode, running with 16-bit addresses, operators, and operands. In other words, 1000 Mhz. Pentiums on startup are emulating a 16-bit, 6 Mhz, near 25-year-old processor. BIOSes even now have to take into account such unpleasant details as near and far pointers (see the PXE standard for some examples).

Some of the new open source BIOS efforts have taken the approach of doing a fair amount of startup in 16-bit assembly, and at some point making a transition to protected mode. There are three serious problems with this approach. First, the 8086 emulation is guaranteed to work for DOS, but we have seen cases where certain instruction sequences don't seem to work right. Emulation has its limitations, and depending on it seems very risky. Second, some hardware initialization absolutely requires having access to 32-bit addresses. For example, setting up PC-100 SDRAM requires at some point a write to all the memory SIMMs, which are 256 Mbytes in some cases. This addressing requires 30-bit addresses to cover 1 Gbyte of RAM. These addresses are not accessible in 16-bit mode, requiring extraordinarily difficult initialization sequences. Finally, use of 16-bit code requires use of a 16-bit assember such as NASM or as86. Having any assembly code is bad, but having two different kinds of assembly code and two assemblers is unacceptable.

Our decision wat to have LinuxBIOS make a transition to 32-bit mode immediately. The transition is actually a fairly simple process: the processor has to load a segment descriptor table (the so-called "global descriptor table" – GDT) and enable memory protection. To load the GDT the processor must execute an LGDT instruction, and supply a pointer to a table descriptor in addressable memory. Fortunately there is no problem with having this table in NVRAM, so moving to protected mode in the first few instructions is no problem. In fact the sequence takes about 10 instructions.

Once the processor is in 32-bit mode it must do basic chipset initialization. While one might expect that chipset initialization would require reams of assembly code, in actuality assembly code is only needed to turn DRAM on. Once DRAM is on, C code can be used. The advantage of using C, besides the obvious improvement in productivity, is that non-Pentium mainboards have a lot in common with Pentium mainboards, and in some cases even use the same chipsets. LinuxBIOS code can be portable between these different mainboards.

Another potential problem is that LinuxBIOS boots and runs on hardware that is completely uninitialized. LinuxBIOS can not make any assumptions about the state of any hardware; the hardware is in an indeterminate state. Linux, on the other hand, assumes that all hardware is initialized by the BIOS. While most Linux hardware initialization still works with uninitialized hardware, we are also finding that we have had to make minor changes to the kernel. For example, Linux assumes that if an IDE controller is not enabled, it is because the BIOS disabled it. On uninitialized hardware, however, the IDE controller is not enabled because there is no BIOS to enable it in the first place. Thus, 'IDE controller not enabled' has a diametrically opposite meaning in BIOS and non-BIOS environments. We have chosen to make the one-line change to the Linux IDE driver to enable IDE controllers when they are found. The change is controlled by an #ifdef LINUXBIOS.

In the end, the structure we arrived at is very simple. There are five major components: protected mode setup; DRAM setup; transition to C; mainboard fixup; and kernel unzip and jump to kernel. We cover these components below.

3.1 Protected mode setup.

Protected mode setup is 17 assembly instructions that put the segmentation, paging and TLB hardware in a sane state and then turns on protected mode (segmentation, not paging). It operates as follows:

1. First instruction, executed in 16-bit mode at address 0xffff0: jump to BIOS startup. This jump is a standard part of x86 processor reset handling.
2. Next five instructions: disable interrupts, clear the TLB, set code and data segments registers to known values.
3. Next instruction: load a pointer to a Global Descriptor Table (GDT). The GDT is a control table for managing addressing in segmented mode.
4. Next four instructions: turn on memory protection
5. Next several instructions: do remaining segment register setup for protected mode
6. At this point, 17 instructions in, we are in protected mode, can address 4 GB of memory, and are running as a Pentium, not an 8086.

3.2 DRAM setup.

DRAM on current mainboards often requires some level of configuration. SDRAM can be particularly tricky. Since there is no working memory hardware initially, this code is assembly. This code has proven to be the hardest code to get working, consuming several weeks on each mainboard. But as we build experience it has also gotten easier to figure out. We resolved several L440GX+ bugs after getting the code working on the Procomm BST1B. This code takes 79 lines of assembly on the Intel and 280 lines of assembly on the Procomm. These numbers will change somewhat as we plan to modify the Intel support to use the Serial Presence Detect capability on the SDRAM.

3.3 Transition to C.

The rest of LinuxBIOS is written in C, so the next few instructions set up the stack and call a function to do the remaining hardware fixup.

3.4 Mainboard fixup.

Mainboard fixup involves limited hardware initialization that we need to finish loading the kernel. In the current implementation, this includes:

- doing whatever the mainboard requires to turn on caching (setting up an MTRR on some processors) so that kernel unzip runs in reasonable time. If the MTRR is not on, the unzip stage can take one minute. With caching on it is not easily measured.
- Making the full FLASH memory available to the processor. Most standard chipsets come up with only 64K or 128K of the FLASH addressable. Enabling all of FLASH requires some register manipulations, which are different on each chipset.
- Enabling minimum device capabilities in the power management hardware (if there is any).

We also need cover things that Linux can't (or won't) do. Linux can not yet initialize a completely uninitialized PCI bus, although it is getting close. We do the bare minimum by setting the Base Address Registers to reasonable values, and Linux properly handles the rest of the work, such as setting up interrupts and turning off the option ROMs. Linux also needs the keyboard to be in some sort of reasonable state, so we reset it. Finally, clock interrupts have to be working, so we make sure these are turned on. This last detail is mainboard-dependent to a limited extent.

Mainboard fixup is currently 330 lines of C code.

3.5 Inflate the kernel.

This is fairly standard code grabbed from the Linux kernel. It has been extended in a few ways. First, parameters from LinuxBIOS to the Linux kernel proper are handled in this code. The parameters are copied to the standard location for Linux kernels for the given architecture. The command line is also copied out. Also, the standard Linux gunzip won't work in a ROM-based environment without some changes. For the most part these involve declaring initialized arrays as 'const' so that they are placed in the read-only text segment (i.e. FLASH) instead of RAM. Initialized automatic and global variables must also be changed so that they are initialized at runtime.

Once parameters (such as memory size) and cmdline are copied out, the standard Linux kernel gunzip is called and the kernel is unzipped to the standard location for the architecture (0x100000on PCs).

3.6 Jump to the kernel.

This is a simple jump instruction. We jump directly to the start of the kernel, since much of the standard Linux kernel setup code is not needed, including the part of the kernel that uncompresses the rest of the kernel; that work is done in the LinuxBIOS startup code. Instead of jumping to the boot setup code as LILO does, we directly enter the kernel in the startup_32 function.

3.7 Summary

The structure of the LinuxBIOS is simple. There is just enough assembly to get things going, and the rest is C. There is just enough C to get the hardware going, and the rest is Linux. For example, we only initialize enough of the MTRR registers (one in most cases) so that the LinuxBIOS code is cached. Linux redoes MTRR initialization anyway: there is no point in doing more than the bare minimum. We do very little PCI configuration, in fact just enough so that Linux can do the rest. Also, LinuxBIOS is configured at build time (somewhat as the BSD kernel is) so that it is configured to the mainboard it is built for. The only code in the mainboard NVRAM is the code needed for that mainboard.

We have been told that 100% of common PC BIOS code is assembly. If we count Linux, then something like 1% of LinuxBIOS is assembly. Because so much of the standard PC BIOS is object code, a lot of code in the standard PC BIOS is directed to figuring out what the hardware is. We do not have this problem, since we are source-based.

4 Status

LinuxBIOS is currently booting from power-up on two mainboards. The first is the Intel L440GX+, and the second is the PROCOMM BST1B. We have just begun work as of June on the Compaq DS10 (Alpha) and Dell 2450 (dual Pentium) mainboards. VIA is sending us a motherboard based on their chipset and we expect to start that work in July.

We generally hear two major concerns about LinuxBIOS. The first is that it is going to be impossible for us to track and support all the various mainboard chipsets in use, which in turn will make coverage of 100% of the mainboards impossible. The second is that less than 100% coverage is equivalent to failure.

It is true that the first two mainboards took a lot of work – about four months of one person for both mainboards. Much of this work was non-recurring, as it involved working out the structure of LinuxBIOS and learning about such things as the quirks of SDRAM and the limits of Linux PCI initialization. We hope to have a better estimate of effort involved once we have completed the next two or three mainboards.

While it is true that we can not hope to cover every mainboard in existence, we do not plan to. Our long-term goal in this effort is to have the vendors pick up the porting effort. One (SiS) is already devoting substantial personnel effort to supporting their new 630 chipset. Other vendors are also interested, once we have developed an initial proof-of-concept for one of their mainboards. One or two other vendors are discussing the construction of mainboards designed from the start to run only LinuxBIOS.

The second concern is coverage. If we do not cover 100% of the mainboard market, have we failed?

We would argue that this question is ill-posed for two reasons. First, coverage at an instant in time says nothing about eventual coverage. In 1991, Linux and the BSD operating systems initially only covered a very small fraction of mainboards extant. Many argued that these systems could never succeed, since they could never support all the hardware available. Linux and the BSDs are very successful, and they still do not cover 100% of the mainboards for sale.

Second, 100% coverage is not necessary for success. If LinuxBIOS is available on a reasonably large number of mainboards, and we can buy those boards for clusters, then for our purposes we have succeeded: the market for LinuxBIOS mainboards is economically viable, and we can buy what we need. Our goal is not "LinuxBIOS on every desktop". Our goal is to be able to buy LinuxBIOS-based mainboards from which to build clusters and other computer systems. Economic viability in this market doesn't require 100% mainboard market penetration.

5 Post-startup Scenarios

In this section we describe some possible uses of the Linux BIOS once it is booted. These uses include a network boot; standard boot; a diskless node boot; and maintenance activities. There are also some unique startup modes that have not been tried to date for clusters. In each case, the Linux kernel can boot another Linux kernel using the LOBOS system call we describe in a companion paper.

5.1 Network boot

Once the BIOS has booted Linux from NVRAM, the kernel take can a number of actions. One would be to establish an SSH connection to a remote DHCP configuration daemon. Using SSH represents a significant advance over the UDP-based approach used by, e.g., bootp and PXE. The SSH-based connection can be much more secure. The SSH connection also benefits from the more stable behavior of TCP under load, as compared to a UDP-based approach. We have seen cases where 32 nodes try to netboot off of one server and only 20 succeed. PXE would have severe problems in this case as the PXE standard suggests that a node only send four packets, then give up.

The DHCP daemon can tell the node its identity and direct it as to which kernel to boot. The DHCP server can even send a kernel image over the SSH connection, and the kernel can boot it using LOBOS.

In order for the kernel to use SSH, we need basic functions that are currently buried in the various SSH client programs. To address this problem we are building a library, based on the OpenSSH source, that will allow both kernels and user programs to create connections to SSH daemons and establish encrypted TCP connections. We may also see if CIPE is useful for this purpose.

5.2 Standard boot

The DHCP daemon could send the kernel its parameters and then tell it to continue booting. In this case, the NVRAM kernel is the kernel of choice; no further kernel loading is needed. Or the kernel could boot another kernel off a local disk or other file system resource such as CDROM. This standard boot is almost identical to what is done on cluster nodes now save for two crucial differences:

1. The boot sequence is entirely under the control of a remote node. The boot will be about as fast, but there is much better control.

2. All existing boot sequences for cluster nodes rely on devices with moving parts, either floppy disks, CDROMs, or hard disks. Our LinuxBIOS-based boot sequence relies on devices with no moving parts, namely the NVRAM on the mainboard. If the node is told to use a hard drive and the hard drive has failed, the node can report the failure to the control node. No more guessing when a cluster node won't come up!

5.3 Diskless node

The DHCP daemon could direct the kernel to mount file systems via NFS (no problem since this is a full-featured kernel), AFS, Coda, InterMezzo, or any other network file system. The kernel could then proceed or boot a different kernel via the network file system being used. The kernel can also use many different transports for the network flie system, such as MyriNet, GigaNET, and SCI. We have worked with Peter Braam to enable InterMezzo to cache an entire root file system, so one option is to cache the root file system to a RAM disk. Experiments have shown that a capable root file system can be contained in a 256 MByte RAM disk. Since InterMezzo supports fetch-on-demand, initially only about 50 MBytes of files need to be loaded.

5.4 Maintenance

The DHCP daemon could also direct the kernel to make an SSH port available for remote maintenance. The node would thus not even start /sbin/init, and would instead wait for instructions from a remote maintenance control program. Instructions could include changing LinuxBIOS parameters or even writing a new test kernel to NVRAM. The kernel could even load a new root file system or repartition the local disk. This maintenance model represents a major advance over what we have now.

Using Linux to write to the NVRAM is tricky. If the write fails for any reason there needs to be a way to recover. For now, we are depending on the BIOS recovery NVRAM. In future, and as the on-board NVRAM grows in size, we expect to have two kernels in the NVRAM, a recovery kernel and a normal boot kernel. The bootstrap code will decide which one to run. If the recovery kernel is damaged in some way the bootstrap code can run the recovery kernel.

Another possibility is to have the kernel open a port and communicate using HTTP commands. We could do BIOS maintenance with a web browser.

5.5 Netboot over Myrinet

All PC netboot standards extant (including PXE) rely on a netboot ROM running in 16-bit mode for network packet I/O. Needless to say this requirement greatly reduces the number of interfaces we can use. Since LinuxBIOS boots a true Linux kernel, we can use high performance network interfaces such as Myrinet for the netboot process. Loading a new disk image over Myrinet would similarly be much faster than Ethernet-based disk image loading.

5.6 Netboot over Scaleable Coherent Interface (SCI)

This model is perhaps the most interesting. SCI is a memory-based network, and moving data requires only a bcopy. When the kernel comes up it could configure an SCI interface. Once the kernel has the interface configured it could use SCI (via a fetch-and-add operation which only takes 6 microseconds) to notify a remote server. The remote server can bcopy a kernel image to the local node, and the local node can use LOBOS to boot this image. Or, more interesting, the remote server can bcopy a whole ramdisk image in, and the local kernel can use that image. SCI bandwidth on 32-bit PCI is about 80 MBytes/second, so moving over a RAMDISK image could be done in a few seconds. SCI has the further advantage that configuring the interface only requires 128 bytes of configuration data, and in fact the interface can be configured from a remote controller node. LinuxBIOS doesn't have to load microcode into the interface, as it does for Myrinet.

5.7 Netboot using IP Multicast

If the whole cluster is booting, we can use IP Multicast to distribute kernel and disk images. Most current tools that support this mode (e.g. Ghost) run under DOS. In our case, we have a full Linux kernel at our disposal and can use IP Multicast for most data, and open TCP sockets as needed to recover lost packets.

6 Current Status

LinuxBIOS is currently booting from power-on to multiuser mode on our Intel L440GX+ and Procomm BST1B mainboards. We are also working with Compaq on the Photon (Pentium) and DS10 (Alpha) systems; Dell on their 2450 server; and VIA on one of their mainboards. The code is available on sourceforget.net.

7 Conclusions

LinuxBIOS is a small BIOS that completely replaces the standard BIOS with a simple bootstrap that loads Linux from NVRAM and starts it up. LinuxBIOS makes new types of cluster configuration, maintenance, and startup models possible that were not practical to date. Based on our near ten years of work with clusters we feel that LinuxBIOS is essential to the construction of maintainable clusters.

LinuxBIOS is working and we are beginning to receive some interest and support from vendors. One vendor has sent a pre-production mainboard and most of their BIOS source. Other vendors are providing information and engineering support, including support for non-Pentium processors. We feel it is only a matter of time before the 16-bit, closed-source model of BIOSes is abandoned in favor of Open Source BIOSes with extended capabilities that the community needs.

LOBOS: (Linux OS Boots OS) Booting a kernel in 32-bit mode

Ron Minnich

August 14, 2000

Advanced Computing Lab, Los Alamos National Labs, Los Alamos, New Mexico

Abstract

LOBOS (Linux Os Boots OS) is a system call that allows a running Linux kernel to boot a new kernel, without leaving 32-bit protected mode and, in particular, without using the BIOS in any way. This capability in turn allows Linux to be used as a network bootstrap program and even as a BIOS, both of which we are working on now. In this paper we discuss how LOBOS works, how we use it, and how LOBOS makes Linux usable as a BIOS, replacing the proprietary PC BIOSes we have today. [1]. LOBOS has been used by two other groups as a reference implementation for their Linux-boots-Linux system calls. One of these other implementations, bootimg, may become a part of the 2.4 kernel.

1 Introduction

At the ACL we have built Linux clusters of 64 nodes, and most recently have built a larger cluster of 128 nodes. While we currently use the 'magic floppy' approach for loading and reloading cluster nodes, we know that this approach will not scale to even 256 nodes – it takes far too much time and effort to put floppies into 256 nodes and make sure they boot properly. We have also found that we need to have absolute control at boot-time of what the node does, even if we are not reloading or initializing the node. We might have half the cluster running a different version of Linux with a different root file system at different times. We might even let jobs in a queueing system indicate which kernel they needed to run, and part of the work of starting a job on a set of cluster nodes might be booting those nodes with the proper kernel.

To support our needs we have decided we will use a netboot-style initialization for both normal operation as well as loading and reloading the cluster nodes. Each time the node is booted we can control which kernel to run, how to get the kernel (over the network or on the disk), and what root partition to use, either local or via NFS, even though in many cases the operating system is booted from the local disk, and the root flie system is chosen from one partition of the local disk.

In this paper we will describe our approach to netboot, which is to use Linux as the bootstrap instead of a special bootstrap program. We first provide an overview of how netboot has been done in the past, how it is being done now in the Windows/PC world, and the problems with the current PC approaches. We close with a disussion of how we might extend our work and use Linux as the BIOS, and hence save a few steps and a lot of time in the booting process.

[1] It's a mere coincidence that many other things in New Mexico are called LOBOS too.

2 Netboot overview

Netbooting has been around in the workstation world for many years, with perhaps the most capable systems being offered by Sun Microsystems. On a Sun system (or, nowadays, any system that runs OpenBoot firmware such as a Power Macintosh), one can simply type 'boot net' and the PROM-based bootstrap code is able to:

1. Initialize the network interface
2. Send out broadcast or point-to-point IP packets to locate a tftp server
3. Load a secondary bootstrap from the tftp server

The secondary bootstrap in turn is capable of mounting NFS partitions, disk partitions, and so on to locate and load the actual kernel to boot. Net booting on Suns has been used for almost 15 years. The protocols are open and there are many open source tftp servers that can support Sun clients for netboot.

In the PC world the situation is not nearly as good. Even today, few PC BIOSes are capable of supporting a netboot option. Even if the BIOS understands netboot, the user often has to procure a PROM for the network card, which of course only works on that one card, and only if the card vendor has provided PROM software. Both the BIOS and the network card PROM are 16-bit 8086 code. As a result, 8086 mode operation is more important than ever. We would like to see 8086 emulation gradually grow less important and eventually disappear, but the netboot standards being promulgated by Intel and Microsoft are leading us the other way.

A further problem is the nature of the standards for netboot. The network card boot model has to conform to a standard interface (NDIS2, a 16-bit Windows model) designed by Microsoft. Intel is working out the BIOS API as well as the network protocols.

As a result of these two trends, PC netboot is going to be 16-bit code cleaving to a network card APIdefined by Windows, using an Intel-defined BIOS API and Intel-defined protocols. Much of this code is proprietary, and using the BIOS for netboot will require us to continue relying on an 8086 assembler. We end up more dependent on 16-bit code running on an emulation of a 20-year-old processor, all of which is proprietary. This is not progress.

2.1 Our requirements for netboot

Given this undesirable situtation we decided to give the problem another look, taking nothing for granted. Our goals are simple: we want to load something onto the CPU that in turn can load boot parameters over the network interface, find out what to do, and then load a kernel. Whatever it is has to be Open Source – we are no longer interested in burning proprietary binaries into PROMs.

We have a few other goals:

1. We don't like assembly code. Also, we have no desire to put a lot of effort into x86 assembly and then repeat our effort with, e.g., Alpha assembly. Therefore, any code we write will be C or better, unless it is impossible to escape assembly.
2. We don't like code that only works for a particular Ethernet card. There are a number of packages for netboot available but their usefulness is strictly limited to a small number of cards. We want to support any network card that Linux supports.
3. We don't see any point in reinventing the wheel. If there is code available that supports lots of network cards, file systems, disk types, and boot protocols, why start from scratch?
4. We don't want to count on the features of any one motherboard. If a motherboard supports netboot, that's no real help, since we don't expect to use that motherboard forever.
5. We want standard protocols, such as NFS, bootp, and so on.

3 The New Netboot

We realized that the requirements for our netboot could be met in one of two ways: we could write a new netboot program from scratch, or we could build a netboot using a minimal Linux kernel. Although there is an apparent advantage to writing our own program from scratch, experience shows that it is not a real advantage. The Sun netboot code has to support many of the same capabilities as a full-blown operating system: it has to be able to do NFS mounts, mount disk partitions, and so on. At the same time, there are many types of file systems it can not use, such as msdos or AFS. Finally, there is no huge savings in space: the network bootstrap is 128 Kbytes. A minimal Linux kernel is 300K. Given the current cost of storage, the difference is insignificant. We decided to go with a minimal Linux kernel for our bootstrap.

3.1 How the new Netboot works

The new netboot works as follows. The netboot code is actually a tiny Linux kernel. It doesn't have much – basically disk, filesystem, network and NFS code. In the current version it does not even need to be able to run user-mode programs – it never exits kernel mode. All it has to do is the following:

1. Boot (eventually from NVRAM, for now from floppy, CDROM, or hard drive)

2. Contact BOOTP server and get parameters for this machine

3. Mount a remote file system via NFS or AFS; or mount the disk or floppy or CDROM.

4. Overlay the currently running kernel with the new file.

Items 1-3 exist in current Linux. The only thing missing is the ability to overlay the kernel with a new kernel. In a sense we need exec for the kernel. The steps required to support this operation are:

1. In kernel mode, open the file and read it into memory. This step is done in kernel mode so that we need not depend on starting */etc/init* and having a user program read the file in. In other words, a kernel can boot a new kernel without even starting any user-mode programs. The file must be read into memory but not into any area of memory occupied by the existing kernel – the existing kernel has to keep running, so overlaying the current kernel code as the file is read in is out of the question. Overlaying the running kernel is the *last* step.

2. Move critical kernel structures into a safe place. These structures must be moved out of the way when the new kernel is copied over the running kernel. So far these structures include Virtual Memory (VM) support structures such as page tables and, on the Pentium, the Global Descriptor Table (GDT); and the parameters used by the kernel when it boots to locate the root partition, as well as any arguments passed to the kernel from the boot command line. These structures will soon also include the log buffer, so that kernel *printk* messages are not lost on reboot.

3. Turn off interrupts. This is the point of no return, so any error checking should have been done by this point.

4. Switch the VM hardware over to the new page tables (and GDT, on the Pentium).

5. Copy the final bootstrap code to a safe place where it will not be overlayed by the new kernel code. The final bootstrap code is simple: it performs a copy of the kernel to the standard location (0x100000), overlaying the currently running kernel.

```
        .long SYMBOL_NAME(sys_ni_syscall)          /*
streams1 */
        .long SYMBOL_NAME(sys_ni_syscall)          /*
streams2 */
        .long SYMBOL_NAME(sys_vfork)         /* 190 */
        .long SYMBOL_NAME(sys_lobos)         /* 191 */
```

Figure 1: Additional system call entry for lobos at 191 in the 2.2.13 kernel

6. Jump to the final bootstrap code. The final bootstrap code copies the new kernel into the right place and jumps to it.

We call this "kernel exec" *LOBOS*, for Linux Os Boots OS. In the next section we discuss its operation in more detail.

3.2 LOBOS implementation

The LOBOS implementation consists of five major pieces, resulting in the addition of 300 or so lines to the kernel. A context diff to apply these changes to a 2.2.13 kernel is available at www.acl.lanl.gov/~rminnich. The basic pieces are as follows:

1. Entry for the *lobos* system call in arch/i386/kernel/entry.S

2. Some additions to the arch/i386/kernel/head.S to make room for the 'safe areas' for the GDT, page tables, kernel startup parameters, and other information

3. The code to read in the new file, in kernel/sys.c

4. The code to turn off interrupts, move the processor page tables and GDT, and switch over to the new page tables and GDT, in arch/i386/kernel/process.c

5. The code to copy the new kernel to the right place and jump to it, in kernel/sys.c

We will go over each of these in turn.

3.2.1 System Call Entry Point

The system call entry point is simply an additional line to arch/i386/kernel/entry.S, as shown in 1

```
/* here begins the support for a kernel rebooting a kernel.
Not all this stuff
 * is used yet. Also, at some point, the logbuffer goes here
so that logs are
 * preserved across reboots
 */
ENTRY(reboot_gdt)
.org 0x7000
ENTRY(reboot_pgdir)
.org 0x8000
ENTRY(reboot_code)
/* leave padding for later use, i.e. a log buffer that survives
reboot*/
.org 0x10000

.globl SYMBOL_NAME(reboot_gdt)
.globl SYMBOL_NAME(reboot_pgdir)
.globl SYMBOL_NAME(reboot_code)
/* end reboot stuff */
```

Figure 2: How the safe areas are declared in head.S

3.2.2 Safe Areas

The safe areas consist of a few additional pages at the beginning of the kernel virtual address space. The lobos bootstrap code knows not to touch these pages, and they are not used in normal kernel operation. Hence this memory represents a safe place to put data that will not be changed by either lobos or the kernel. Currently the GDT, reboot code, and kernel parameters are saved here. The code for the safe areas is shown in 2. The reboot_pgdir area is not currently used.

3.2.3 Reading in the file

The real meat of this system call is the work done to read in a file and set the kernel up for reboot. This work occurs in a few places. The first is the sys_lobos system call, which we show in 3. This function is called with a name. It first gets a copy of the file name via getname, then performs a lookup on the file.

The function has to get access to a file, which is done via the *lookup_dentry* call. We double check to make sure there is a real inode associated with the dentry, although this level of checking is probably unnecessary. The size of the file is contained in the inode structure. We allocate that amount of memory and, if the allocation succeeds, call the kernel *read_exec* function to actually read the file into

```
/* get a dentry via lookup, then use the open_private func-
tion to open
 * it, then use read_exec to read it.
 */
asmlinkage int sys_lobos(char *file)
{
 char *name;
 struct dentry *d;
 name = getname(file);
 printk("sys_bootfile: file ptr is %p\n", file);
 if (! name)
   return -EFAULT;
 printk("the name is %s\n", name);
 d = lookup_dentry(name, 0, 1 /* read only */);
 if (d)
   {
     void *v;
     int result;
     int good = 1;
     size_t count;
     printk("good open, dentry is %p\n", d);
     if (! d->d_inode)
       good = 0;
     if (! good) printk("NO INODE!\n");
     if (good) {
       count = d->d_inode->i_size;
       printk("the size is %d\n", count);
       printk("let's try to mallo that much\n");
       v = vmalloc(count);
       if (v) {
         result = read_exec(d, 0, v, count, 1);
         printk("read result is %d\n", result);
         if (result == count)
           run_boot_file(v, count);
       }
       else printk("alloc failed\n");
     }
   }
 else
   printk("open failed, d is null\n");
 return -EINVAL;
}
```

Figure 3: Top level of the sys_lobos system call

memory. Although *read_exec* is intended for reading in executable files, it also serves perfectly for our purposes.

At this point much of the work is done. The final steps are handled by the function *run_boot_file*, which is called with a pointer to the kernel area and a size. This function is shown in 4.

This function copies the final bootstrap, *do_boot_file*, to the safe memory location. It calls *os_restart* to set up the virtual memory structures (GDT and page tables on the Pentium), and finally calls the final bootstrap code to the do actual final step of copying the new kernel over the current kernel. If anything fails, the current behaviour is to hang forever, although obviously the correct long-term behavior is to reset the machine.

3.2.4 Setting up the page tables and GDT

This work is done by the *os_restart* function. This function has to change the state of the virtual memory hardware and by its very nature represents the most machine-dependent code in LOBOS (the assembly code presented above for reserving space and system call table entries could just as easily be C code, and is in many kernels).

The main goal of this function is to move the GDT and page tables out of the way, and to do it in a way that allows the VM hardware to function until the new kernel takes over and loads the hardware with the new kernel's GDT and page tables. Currently, the GDT is put in the safe area, and the page tables are put in an area allocated in high memory. We use the allocated memory for the page tables as they can vary in size for different types of processor. Pentium-compatible processors that support 4 MByte page table entries only need one page to address 4 Gbytes of memory; processors that only support 4 Kbyte page table entries need much more space.

The steps here are as follows: store the current gdt into curgdt, so we can find out where it is. Get the pointer to the safe gdt, and copy the first page of the current gdt to it. We only need a very small part of the GDT, but for now we just grab the whole first page. Next we allocate a new page table and copy the current page table to it. Note that for now this code only works for machines with 4 MB page table entries. Next we switch to the new GDT (the sgdt instruction); and finally we switch to the new page tables. At this point the kernel can be safely overwritten by the final bootstrap. The only assembly code in this function is for very low-level hardware support.

```
void run_boot_file(void *kernel, size_t count)
{
 extern void os_restart(int);
 extern char saved_bootparams[4096];
 extern void *reboot_code;
 int result;
 unsigned long *test = kernel;
 void *setup = 0, *kernelstart, *bootsector = 0;
 size_t funcsize = ((unsigned long) end_boot_file) -
   ((unsigned long) do_boot_file);
 void *v;
 typedef int (*z)(void *v, size_t count, void *setup, void
*kernelstart,
            void *bootsector, int testonly);
 z bf;
 cli();
 kernelstart = __va(0x100000);
 v = &reboot_code;
 /* copy it out */
 memcpy(v, do_boot_file, funcsize);
 os_restart(0);
 /* copy out saved_bootparams ..*/
 printk("copying out ssaved_bootparams\n");
 memcpy(__va(0x90000), saved_bootparams, 4096);
 /* now call it */
 printk("allocated %d bytes, now call %p\n", funcsize, v);
 bf = v;
 result = (*bf)(kernel, count, setup, kernelstart, bootsec-
tor, 0);
 printk("RETURNED FROM do_boot_file: HANGING
FOREVER\n");
 while(1);
}
```

Figure 4: The run_boot_file function.

```
void os_restart(int notused)
{
 void *newgdt = 0;
 extern char *reboot_gdt;
 pgd_t *newpagedir = 0;
 unsigned long cp;
 void *gdtbase;
 int gdtsize;
 unsigned long l;
 unsigned long curpagetable;
 unsigned long x;
 int i;
 printk("os_restart ...\n");
 curgdt[0] = curgdt[1] = 0;
 __asm__ __volatile__ ("sgdt %0" : "=m" (curgdt));
 newgdt = & reboot_gdt;
 gdtsize = 4095;
 memcpy(newgdt, gdtbase, gdtsize + 1);
 /* build the new page dir that is out of the way ... */
 newpagedir = get_pgd_slow();
 if (! newpagedir) {
      printk("newpagedir allocate failed\n");
      return;
 }
 memcpy(newpagedir ,
swapper_pg_dir, sizeof(swapper_pg_dir))
 l = (unsigned long) newgdt;
 curgdt[1] = l >> 16;
 curgdt[0] = ((l & 0xffff) << 16) | gdtsize;
 cli();
 __asm__ __volatile__ ("lgdt curgdt");
 __asm__ __volatile__ ("ljmp $0x10,
 $blahblah\nblahblah:nop\n");
 __asm__ __volatile__(
                "movl $0x18,%eax\n"
                "movl %eax,%ds\n"
                "movl %eax,%es\n"
                "movl %eax,%fs\n"
                "movl %eax,%gs\n"
                "movl %eax,%ss\n"
                );
 SET_PAGE_DIR(current,newpagedir);
 return;
}
```

Figure 5: os_restart code

```
void
do_boot_file(void *v, size_t count, void *kernelstart, int
testonly)
{
 int i;
 void (*f)(void) = kernelstart;
 extern char *reboot_gdt, *get_options;
 volatile unsigned char *src = (char *) v;
 volatile unsigned char *dst = (char *) kernelstart;
 unsigned long *l;

 for(i = 0; i < count; i++, src++, dst++)
   {
    if ((dst >= &reboot_gdt) && (dst < &get_options)) {
      continue;
    }
    if (testonly) {
    }
   else {
          *dst = *src;
      }
   }
 if (testonly)
   return;
 f();
}
```

Figure 6: Final bootstrap, do_boot_file

3.2.5 Final Bootstrap

The final bootstrap copies the new kernel over the old one, skipping the safe areas.

This is a rather simple function, in essence a memcpy. The one difference is that it does not overly the region between *restart_gdt* and *get_options* (the safe area) with the new kernel. Once it has done the copy it calls the new kernel.

3.3 Calling LOBOS from user mode

The program that uses the system call is shown in 7. The program is quite minimal. It takes the name of the file and calls the system call with that name as a parameter.

```
#include <stdio.h>
#include <errno.h>
#include <syscall.h>

#define __NR_bootfile 191

_syscall1(int, bootfile, char *, name);

int
main(int argc, char *argv[])
{
  char *name = "test";
  if (argc > 1)
    name = argv[1];
  printf("name is %p\n", name);
  bootfile(name);
}
```

Figure 7: The bootfile program

3.4 The LOBOS command

Following the model of fastboot(1), we have created a command called lobos. Lobos puts a binary, uncompressed kernel image in /tmp, and creates a file called /lobos. We have modified the reboot script so that if the /lobos file exists, the bootfile program is invoked with the uncompressed kernel image as the argument.

To reboot any kernel, the user can type the full kernel path, or simply the intermediate part of the name, e.g. the command *'lobos linux-2.2.13'* will reboot the the kernel /usr/src/linux-2.2.13/vmlinux.

4 Performance and usability.

Booting a kernel via LOBOS is much faster and easier than the standard BIOS-based boot. There is no long wait common with BIOS boots. The unnecessary memory test and zero is a thing of the past, as is the wait for the many unnecessary tasks that exist only to support DOS 1.0.

We now have a log buffer that survives reboots and that has proven to be a major plus. We much prefer this style of booting to the 16-bit BIOS-based style used on PCs to date.

5 Related work

There are two other systems which allow Linux to boot Linux: bootimg, from Werner Almesberger; and Two Kernel Monte, from Scyld Computing. They differ in philosophy from LOBOS in a few significant ways.

5.1 Bootimg

Bootimg allows a user-mode program to reboot the kernel with a new image. The user program has to read the file into memory, and then calls the bootimg system call. The kernel code is responsible for parsing the header of the file and unzipping the code.

Bootimg turns virtual memory (i.e. paging) off at some point, but leaves i386-style segments on. Turning VM off complicates a number of issues. Since the user buffer is in virtual memory, bootimg must first copy it in to physically contiguous kernel memory that can be addressed with VM off. Also, in the future kernel components may not all be in phsyically contiguous memory; we certainly do not want to count on it. Finally, on systems such as Alpha, turning off VM is tantamount to turning off all protection. Given that even the lowest-level NVRAM software on the Alpha runs with VM enabled, we are worried about any approach that involves turning VM off.

Bootimg constitutes about 1100 lines of code, of which at least 600 are architecture-dependent. There are only 40 or so lines of assembly. One issue is that bootimg does define a number of structures (such as GDTs) that need to maintained in synch with the kernel.

Bootimg can be used with the LinuxBIOS.

5.2 Two Kernel Monte

Two kernel monte (TKM) takes a very different approach to the problem. TKM at some point turns off BOTH VM (paging) AND i386-style segmentation. In order to avoid copies and the requirement for a large area of physically contiguous memory, TKM builds an internal virtual-to-physical page map so that when VM is off, TKM can still get to the new kernel image. Also, once real mode is off, TKM can call the BIOS to reset hardware that may not work properly after a reboot. TKM can not work with the LinuxBIOS, since it depends on the BIOS for a critical part of the reboot step.

5.3 Summary of the three systems

TKM is probably the most architecture-dependent of the three, and LOBOS is probably the least architecture-dependent. LOBOS is less than half the size of the others, and has only a fraction as much assembly code. Bootimg is the most polished in certain ways: it does the most thorough permission checking and ramdisk support, for example. TKM will probably work with just about any kind of video hardware, since it calls the video bios to reset the video card. TKM will probably never work with the LinuxBIOS.

All three systems deal with the problem of VM in very different ways. LOBOS keeps paging and segmentation turned on, and relies on the presence of the "safe areas" to maintain through the reboot process. LOBOS needs to relocate the GDT and page tables once. Bootimg turns VM off, and relies on the presence of physically contiguous memory in kernel mode to get around the lack of VM. Bootimg relocates the GDT four times during a boot. TKM turns VM off and relies on its own virtual-to-physical map to keep track of memory. TKM reloads the GDT once. We feel most comfortable with keeping VM turned on at all times, especially as we move to the Alpha, where there is not support for segmentation.

TKM and Bootimg require an external program to load the image. LOBOS does not; we supply such a program, but a LOBOS-equipped kernel can, given a file name, boot that file. When the LinuxBIOS boots from NVRAM, it can further boot a different kernel (e.g. at the direction of a DHCP server) without ever having to run a user-mode program.

Only LOBOS allows the kernel log buffer to survive across reboots. We have found this capability very useful, since we no longer need to wait for klogd to clean up before rebooting. We are trying to reach a 3-second reboot time, and the fewer processes we have to wait for when we reboot, the better.

In terms of security, all three systems are no more (and no less) secure than a standard reboot system call.

There is a final question: which of these systems will make it as the "standard" in the Linux kernel? While we prefer the LOBOS implementation, and especially some of the LOBOS design decisions, we believe that the standard system call for 2.4 will be bootimg. At some point we will then need to revisit portability issues, since bootimg depends very heavily (over 30% of the code, as opposed to several tens of lines in LOBOS) on aspects of i386 Linux that do not exist in other architectures.

For our own purposes we will probably continue using LOBOS. The two determinants are the ability to boot a new kernel entirely from the kernel, and the fact that the log buffer is preserved across reboots. LOBOS has also demonstrated portability across a wide range of kernels due to its simplicity.

6 Next Steps

We are working on putting a LOBOS-enabled kernel into the FLASH RAM on our Intel 440GX motherboards. We are using code from the OpenBIOS project to bootstrap our kernel into memory. The kernel we boot serves as a true network bootstrap, in that it comes up and asks a manager node what it should do, which may include simply booting from the disk. We report on the new BIOS work in a companion paper.

Our work on LOBOS has been used by other researchers. Werner Almesberger has developed bootimg, which will probably appear in the 2.4 kernel. Researchers at Scyld Computing had started a project similar to LOBOS but had gotten stuck; they were able to use our work to finish their system, Two Kernel Monte.

7 Conclusions

LOBOS is a system call that allows a running kernel to boot another kernel. Once a kernel is running it has no need to use the BIOS to boot other kernels. This new capability allows us to use Linux kernels as a network bootstrap, as opposed to using a special network bootstrap program. It is also very easy to boot new kernels: we simply type in 'lobos <kernel-name>' and the new kernel is up and running in less than a minute. We don't really need LILO any more.

LOBOS also makes it possible to replace the BIOS with a Linux-based BIOS. The benefits to our work are clear: the BIOS is the last great barrier to truly Open Source-based clusters.The BIOS also represents a major stumbling block to managing large clusters, due to its primitive structure and limited capabilities, as well as to its 16-bit unprotected-mode origins. We feel that LOBOS represents a first step to freeing Linux users from the BIOS and all its constraints.

LOBOS has to date been used as a reference by two other groups to build working system calls with similar capabilities. One of these system calls, bootimg, will probably be a part of the standard 2.4 kernel.

KLAT2's Flat Neighborhood Network

H. G. Dietz and T. I. Mattox
Electrical Engineering Department, University of Kentucky, Lexington, KY 40506-0046

Abstract

KLAT2, Kentucky Linux Athlon Testbed 2, is a cluster of 64 (plus two "hot spare") 700MHz AMD Athlon PCs. The raw compute speed of the processors justifies calling the system a supercomputer, but these fast nodes must be mated with a high-performance network in order to achieve the balance needed to obtain speed-up on real applications. Usually, cluster networks are built by combining the fastest available NICs and switching fabric, making the network expensive. Instead, KLAT2 uses a novel "Flat Neighborhood" network topology that was designed by a genetic algorithm (GA). A total of about $8,100 worth of 100Mb/s Fast Ethernet NICs, switches, and Cat5 cable, allows KLAT2's network to deliver both single-switch latency for any point-to-point communication and up to 25.6Gb/s bisection bandwidth. This paper describes how this new network architecture was derived, how it is used, and how it performs.

1. Whence KLAT2?

Since February 1994, when we built the first parallel-processing Linux PC cluster, we have been very aggressively pursuing any hardware and software system technologies that can improve the performance or give new capabilities to cluster supercomputers. Thus, when AMD released the first PC processors with vector floating point support (the K6-2's single-precision *3DNow!* instruction set), we quickly developed compiler technology to support the use of these multimedia-oriented instructions for general vector/SIMD parallel computing. The result has been a strong relationship with AMD in which they have supported our research work in a variety of ways. Most recently, AMD donated sixty-six 700MHz Athlon processors to our work.

We have used these processors to build KLAT2 (Kentucky Linux Athlon Testbed 2). The name is a reference to the advanced alien who came to the earth in *The Day The Earth Stood Still* to warn the people of the earth that if they did not immediately stop fighting among themselves, the planet would be destroyed; the earth's destruction is narrowly averted with the famous command to Klaatu's robot: "Gort, Klaatu Barada Nikto!" KLAT2, Gort, and Klaatu are shown in the photo.

Our goal for KLAT2 was to build a system that can coordinate the Athlons well enough to reach the performance range seen in the list of the "Top 500" supercomputers (**`http://www.top500.org/`**). In fact, KLAT2's single-precision LINPACK benchmark performance would rank it 197th in the June 7, 2000 list.

1.1. Obvious Networks Are Barada

There never was any doubt that Athlon PCs would be very capable supercomputer nodes. However, it was not clear how we could create a cluster network that would balance that performance; AMD only donated Athlon modules — our meager budget would have to cover the cost of turning the processors into complete systems and networking them together. The cost of the motherboards, memory, and cases were all reasonable, but what about the network?

PAPERS (**`http://aggregate.org/`**) is a low-latency network that we have developed through 18 generations of custom hardware since 1994. It is cheap enough for use in KLAT2. However, PAPERS only solves part of the network problem. Although the 3us latency PAPERS achieves is impressive, the aggregate function communication that PAPERS provides is not designed to send blocks of data from one PC to another; a different network is needed for high-bandwidth messaging.

We tried approaching several makers of Gb/s network technologies for donations and/or discounts. However, both the donations and discounts that we were offered were insufficient to satisfy our requirements. The cheapest of the Gb/s NICs that we found were PCI Ethernet cards priced under $300 each, but even that cost would have stretched our budget. Adding to that the cost of a hierarchy of Gb/s switches brings any solution based on Gb/s NICs over $2,000 per PC connected. Further, the switch hierarchy multiplies latency and a tree topology dramatically limits bisection bandwidth. We needed a new solution.

1.2. The New Approach

When no solution seems to work, it is time to rephrase the problem. We wanted to have the minimum possible latency between any pair of PCs. Clearly, you couldn't put 65 NICs in each machine to implement a direct connection... the next best thing would be to have just one switch delay between any two PCs. The problem then becomes that a 66-way switch that can handle communication at full wire-speed is not cheap.

You can buy a wire-speed 32-way 100Mb/s switch for about $525. Thus, we could use 32 dual-processor PCs and channel bonding of multiple NICs (`http://www.beowulf.org/`). Although dual-processor PCs using Intel processors are competitively priced, they divide the node memory bandwidth between the two processors, delivering significantly lower performance than two uniprocessor PCs would for many codes. Even if we wanted to adopt that solution, dual-Athlon PCs are not yet widely available.

The "Flat Neighborhood" network topology came from the realization that it was sufficient to share at least one switch with each PC — *all PCs do not have to share the same switch.* A switch defines a local network neighborhood, or subnet. If a PC has several NICs, it can belong to several neighborhoods. For two PCs to communicate directly, they simply use NICs that are in a neighborhood that the two PCs have in common. Coincidentally, this flat, interleaved, arrangement of the switches results in unusually high bisection bandwidth — approaching the same bisection bandwidth that we would have gotten if we had wire-speed switches that were wide enough to span the entire cluster! We even get the benefit that, because four NICs are available for simultaneous use in each PC, we bypass some of the I/O serialization that using IP would imply with a single Gb/s NIC (or channel-bonded set of NICs) under Linux.

1.3. No Free Lunch

Unfortunately, flat neighborhood networks introduce several interesting new problems. Using KLAT2's network as the primary example, the remainder of this paper discusses:

• How to design a flat neighborhood network. Unfortunately, only very small flat neighborhood network wiring patterns can be designed by hand. We created a genetic algorithm (GA) that can search for an appropriate wiring pattern, also optimizing secondary properties of the network for specific types of communication traffic.

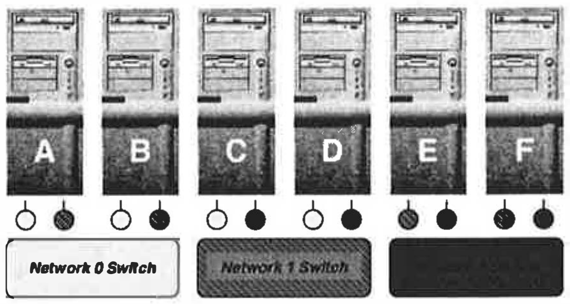
• How to physically wire the network. This may seem like a trivial concern, but flat neighborhood designs do not necessarily have good wiring locality properties and, in the general case, are not regular (i.e., often have no symmetry).

• How to perform basic routing between PCs. Most network hardware and software assumes a variety of network properties that flat neighborhood networks violate. For example, if you ask PC #0 for the network address of PC #1, you do not get the same answer that you get if you ask PC #2 the same question.

• How to take full advantage of extra bandwidth that is available for some (but not all) communication paths. What is needed is very similar to channel bonding, however, the standard Linux support for channel bonding works in a way that is incompatible with the flat neighborhood topology.

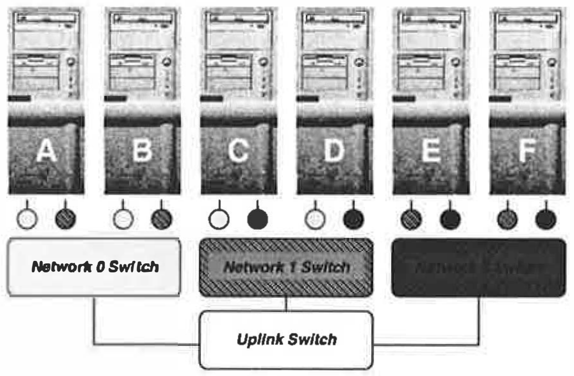
2. The GA Network Design Process

Conceptually, it is very simple to design a flat neighborhood network wiring pattern. For example, for 6 PCs, each with 2 NICs, using 4-way switches:



Thus, A and B are both connected to switches 0 and 1, C and D to switches 0 and 2, and E and F to switches 1 and 2. For A to send to C, it uses switch 0. For A to send to B, it can use either switch 0, switch 1, or both.

In practice, it is useful to reserve one port on each switch for connection to an "uplink switch." This switch is not used in normal operation of the cluster, but provides a very efficient means for communication between the cluster and other systems as well good support for broadcast/multicast. Thus, the above FNN would really be built using 5-way switches and wired as:



Although the use of an uplink switch does not complicate the design problem, the complexity of the design problem does explode when a larger system is being designed with additional, secondary, optimization criteria (such as maximizing the number of switches shared by PCs that communicate in various patterns). Because this search space is very large and the optimization criteria are very general (often requiring simulation of each potential network design), use of a genetic search algorithm is much more effective than other means. The complete GA network design process is described in [1]. Basically, the current version of our GA search uses:

• A specification of how many PCs, the maximum number of NICs per PC (all PCs do not have to have the same number of NICs!), and a list of available switches specified by their width (number of ports available per switch). Additional dummy NICs and/or switches are automatically created within the program to allow uneven use of real NICs/switch ports (e.g., KLAT2's current network uses only 8 of 31 ports on one of its switches).

• A designer-supplied evaluation function that returns a quality value derived by analysis of specific communication patterns and other performance measures. This function also marks problem spots in the proposed network configuration so that they can be preferentially changed in the GA process.

• A crossover operation based on exchange of closed sets of connections between two parent network configurations. The closure operation ensures that the new configuration always satisfies the designer-specified constraints on the number of NICs/PC and the number of switch ports for each switch.

• Several different mutation operators.

• A two-phase GA scheme in which large network design problems with complex evaluation functions are first converted into smaller problems with an evaluation function that weights only the basic flat neighborhood property: having at least one switch shared by each pair of PCs. A number of generations after finding a solution to the simplified network design problem, the population of network designs is scaled back to the original problem size, and the GA resumes with the designer-specified evaluation function.

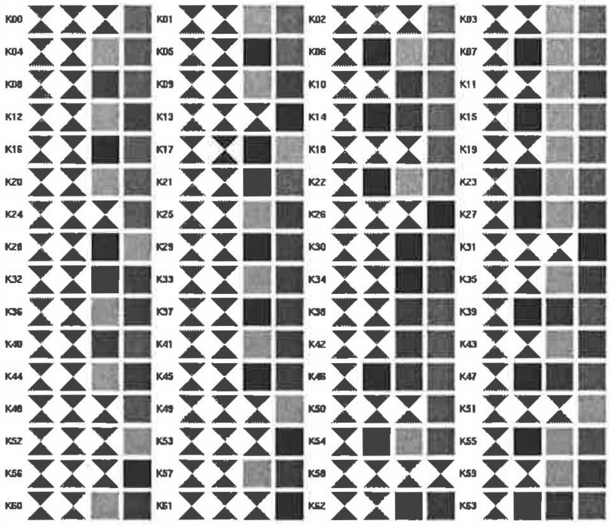
The GA program, written in C, uses SIMD-within-a-register parallelism when executed on a single processor system, but also can be executed in parallel using a cluster. KLAT2's current network design was actually created using our first Athlon cluster, Odie — four 600MHz PCs.

3. Wiring The Physical Network

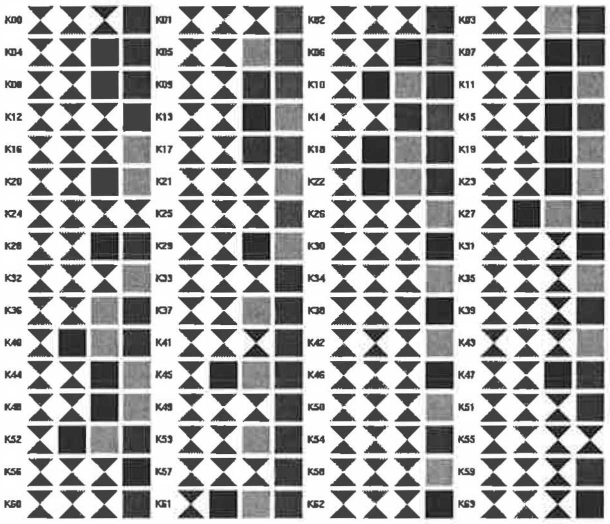
One of the worst features of flat neighborhood networks is that they are physically difficult to wire. This is because, by design, they are irregular and have very poor physical locality between switches and NICs. The GA design process could be made to include a model of physical wiring locality/complexity in its search, but the resulting designs would probably sacrifice some performance in return for the reduction in wiring difficulty.

KLAT2's PCs are housed in four standard 48"x18"x72" shelving units. The network for KLAT2 consists of ten rack-mounted 32-way (really 31-way plus one uplink port) wire-speed 100Mb/s Fast Ethernet switches. Nine of these switches form the flat neighborhood network's switching fabric; the tenth is used exclusively for (1) I/O to other clusters, (2) multicast, and (3) connection of the two "hot spare" Athlon PCs. Thus, KLAT2's network connects 264 NICs and ten switches in a seemingly random pattern spanning five physical racks. Worse still, because we were initially missing some network hardware, we actually implemented one wiring pattern and then rewired the system when the rest of the network hardware arrived.

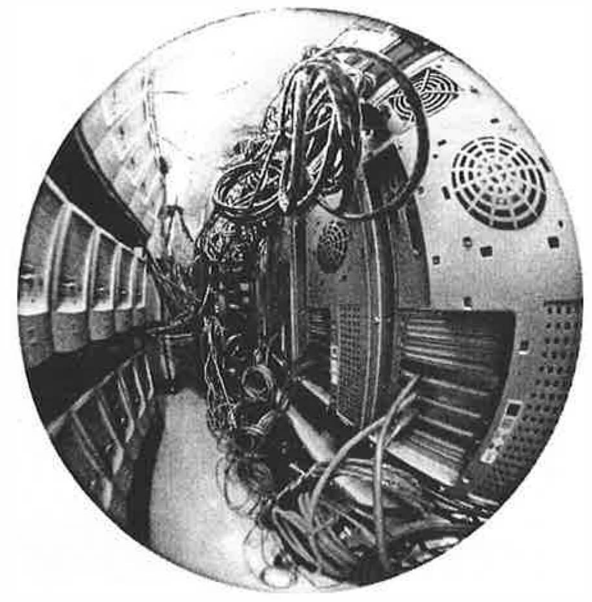
So, how did we physically wire KLAT2? The basic trick involves recognition of the fact that it doesn't mater which switch port each NIC is connected to. Each of the ten switches was assigned a color that was used for all the cables connected to that switch. For each of the PCs, we simply had our GA print out the network design in the form of a label for each PC showing the cable colors that should be plugged into that PC's NICs:



Wiring the cluster with the GA design took no more than a few minutes per PC, including the time to neatly route the wires between the PC and the switches. In fact, we re-ran the GA to optimize for the communication patterns of the DNSTool CFD (Computational Fluid Dynamics) code [4] and physically re-wired the entire system in less than a few hours. The new design is:



The original physical wiring pattern looks like:



4. Basic Routing

The fundamental problems in flat neighborhood network routing are conceptually simple, but break many assumptions made by network software — especially IP-based software. In terms of KLAT2's network in particular:

• Each PC has four different cluster-local addresses, each on a different one of the nine cluster-internal subnets. (A fifth address is aliased with one of the four local addresses for each PC to simplify references made from outside the cluster.) However, because all nine subnets are connected by a tenth switch for multicast, etc., any pair of PCs can communicate through the switches using any NIC in one PC to reach any NIC in the other. The result is that normal route discovery procedures find that everything can reach everything, and thus construct routing tables that do not reflect the flat neighborhood topology. These incorrect routing tables yield terrible performance due to unnecessary, heavily congested, use of the uplinks to the tenth switch.

• Many network libraries, such as LAM MPI [2][3] (**`http://www.mpi.nd.edu/lam/`**), assume that there is only a single IP/MAC address for each node — and that a single server can distribute these addresses. However, this yields essentially the same performance problem discussed above; three of four NICs in each PC would be entirely ignored.

Our basic routing solution is fairly straightforward. Briefly, we have:

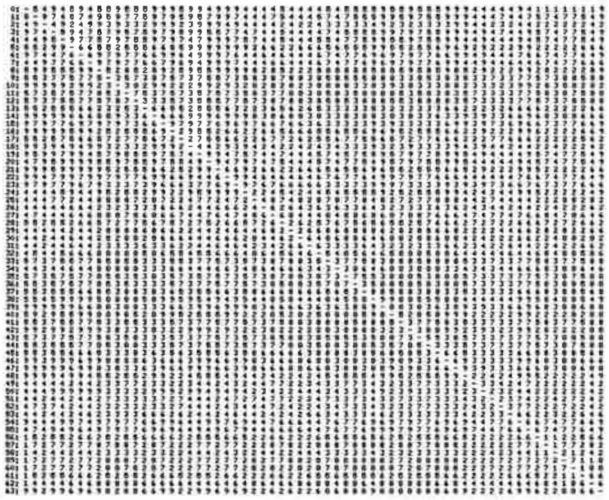
• Created a simple power-on technique for a server to discover which NIC MAC addresses are in each PC.

• Augmented our GA network design software to automatically create a full set of routing tables. Each PC has its own, unique, routing table.

• Taken steps to ensure that no machine will ever broadcast an address resolution request.

• Modified software, such as LAM MPI, so that the customized local routing tables can be used.

A brief summary of the GA-generated basic routing tables follows. Each row corresponds to a particular PC's routing table; each entry specifies which of the nine neighborhood subnets would be used to communicate with the corresponding other PC.



5. Advanced Routing

The basic routing concern is to ensure that communications between a pair of machines go through a single switch; however, many pairs of machines have multiple choices for single-switch connections. For example, KLAT2's nodes k00 and k01 actually have three switches in common, not just one. Thus, a technique resembling channel bonding, but much more complex, can be used to provide up to three times the single-link bandwidth between k00 and k01. In fact, using either PCs or the tenth switch, we can get four times the single-link bandwidth between k00 and k01, although latency on one of the paths will be significantly higher than on the other three.

At this writing, we have determined a reasonable implementation technique for advanced routing (see [1]), but have not yet experimented with it. The technique involves encoding all the viable paths and using a modified lookup procedure to determine the set of paths to use; it ignores any paths that would pass through PCs (i.e., does not use PCs as routers). We intend to add this new lookup procedure to an active message library that we will be building for KLAT2's RealTek-based Fast Ethernet NICs.

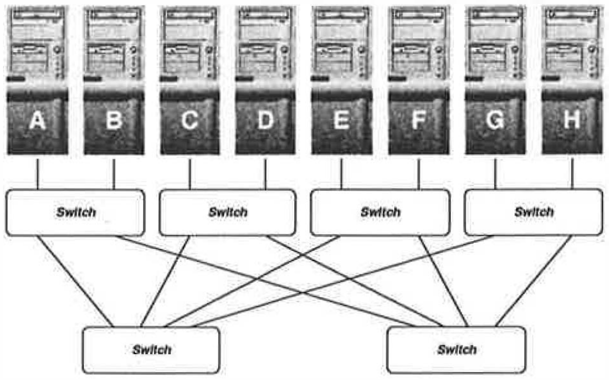
6. FNNs vs. Other Network Designs

To better understand the differences between FNNs and other network architectures, it is useful to perform side-by-side comparisons. The first comparison is based purely on the performance predicted using a relatively simple model. The second comparison uses a variety of standard benchmarks that were executed using KLAT2 with two different routing rules: one using the basic FNN routing, the other treating the FNN with its uplink switch as a tree.

6.1. Predicted Performance

Since large FNNs in general, and the one used in KLAT2 in particular, lack symmetry that would facilitate closed-form analysis, it is most effective to begin with analysis of a symmetric network for a smaller system. Thus, let us first consider interconnection network designs that can be built using four-way switches for an eight-PC cluster.

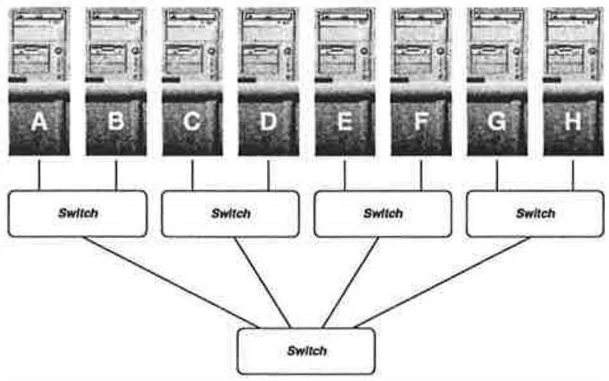
Among the many different topologies proposed for interconnection networks, fat trees have become very popular because they easily provide the full bisection bandwidth. Assuming that appropriate switches are available, the eight-PC network would like like:



For this fat tree, the bandwidth available between any pair of PCs is precisely that of one link; thus, we say that the pairwise bandwidth is 1.0 link bandwidth units. The bisection bandwidth of a network is determined by dividing the machine in half in the worst way possible and measuring the maximum bandwidth between the halves. Because the network is symmetric, it can be cut arbitrarily in half; the bisection bandwidth is maximal when all the processors in each half are sending in a permutation pattern to the processors in the other half. Thus, assuming that all links are bidirectional, the bisection bandwidth is 8*1.0 or 8.0.

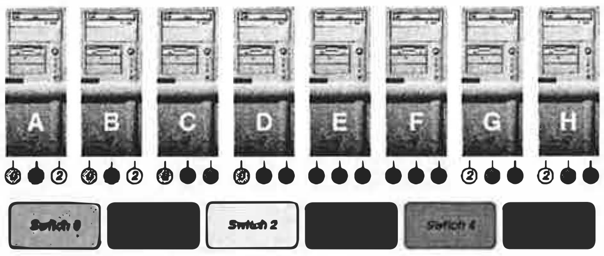
Pairwise latency also is an important figure of merit. Latency introduced by the software interface and NIC is essentially independent of interconnection topology, thus we can ignore this factor when comparing alternatives. In addition, if the cluster nodes are physically near each other, we can ignore the wire latency and simply count the average number of switches a message must pass through. Although some paths have only a single switch latency, e.g. between A and B, most paths pass through three switches. More precisely, from a given node, only 1 of each of the 7 other nodes can be reached with a single-switch latency. Thus, 1/7 of all pairs will have 1.0 switch latency and 6/7 will have 3.0 switch latency; the resulting average is (1.0 + 3.0*6)/7, or 2.7 switch latency units.

Unfortunately, most inexpensive switches cannot handle routing for a fat tree topology. The problem lies in the fact that the switches within the fat tree must be able to balance load using multiple paths between the same two network addresses or subnets. Thus, if switch cost is to be kept competitive, the fat tree arrangement generally is not viable for technologies like Fast Ethernet. In fact, the best conventional topology that most commodity switches were designed to support is a simple tree, such as:



It is not difficult to see that latency of the tree is very similar to that of the fat tree. For some communication patterns, including all those in which only a single PC pair are communicating, the tree does yield 1.0 link bandwidth units for communication between a PC pair. More often in parallel programs, only a fraction of link bandwidth is available because multiple pairwise communications are sharing the bandwidth of the 4 links to the top-level switch. The bisection bandwidth is thus approximately half that of the fat tree. The primary advantage is simply the ability to use dumber, cheaper, switches; the primary disadvantage is poorer performance.

Now consider using a FNN to connect these same eight PCs with four-way switches. Like the tree configuration, the FNN easily can use cheap, dumb, switches that could not implement fat tree routing; in fact, the FNN does not connect switches to switches, so the routing requirements imposed on each switch are minimal. However, more NICs are needed than for the fat tree. At least for 100Mb/s Ethernet, the cost savings in using dumber switches more than compensates for the larger number of NICs. For our example, each PC must have 3 NICs connected in a configuration similar to that shown by the switch numbers and colors in:



Unlike the fat tree, the FNN pairwise bandwidth is not the same for all pairs. For example, there are 3.0 link bandwidth units between A and B, but only 1.0 between A and C. Although the FNN shown has some symmetry, FNN connection patterns in general do not have any basic symmetry that could be used to simplify the computation of pairwise bandwidth. However, no PC has two NICs connected to the same switch, so the number of ways in which a pair of connections through an S-port switch can be selected is S*(S-1)/2. Similarly, if there are P PCs, the number of pairs of PCs is P*(P-1)/2. If we sum the number of connections possible through all switches and divide that sum by the number of PC pairs, we have a tight upper bound on the average number of links between a PC pair. Because both the numerator and denominator of this fraction are divided by 2, the formula can be simplified by multiplying all terms by 2. In other words, the pairwise bandwidth for the above FNN is ((4*3)*6)/(8*7), or an average of about 1.28571428 links.

Not only does the average pairwise bandwidth of the FNN beat that of the fat tree, but the bisection bandwidth also is greater. Bisection bandwidth of a FNN is difficult to compute because the definition of bisection bandwidth does not specify which pairwise communications to use, but the choice of pairings can dramatically alter the value achieved. Clearly, the best-case bisection bandwidth is the number of links times the number of processors; 8*3.0 or 24.0 in our case. A conservative bound can be computed as the number of processors times the average pairwise bandwidth; 8*1.28571428 or 10.28571428. Either of these numbers is significantly better than the fat tree's 8.0.

Even more impressive is the FNN design's pairwise latency: 1.0 as compared with 2.7 for the fat tree. No switch is connected to another, so only a single switch latency is imposed on any communication.

However, the biggest surprise is in the scaling. Suppose that we replace the six 4-way switches and eight PCs with six 32-way switches and 64 PCs? Simply scaling the FNN wiring pattern yields pairwise bandwidth of ((32*31)*6)/(64*63) or 1.47619047, significantly better than the 8 PC value of 1.28571428. FNN bisection bandwidth increases relative to fat tree performance by the same effect. Although average fat tree latency decreases from 2.7 to 2.5 with this scaling, it still cannot match the FNN's unchanging 1.0.

It also is possible to incrementally scale the FNN design in another dimension -- by adding more NICs to each PC. Until the PCs run out of free slots for NICs, bandwidth can be increased with linear cost by simply adding more NICs and switches with an appropriate FNN wiring pattern. This is a far more flexible and cheaper process than adding bandwidth to a fat tree.

6.2. Measured Performance

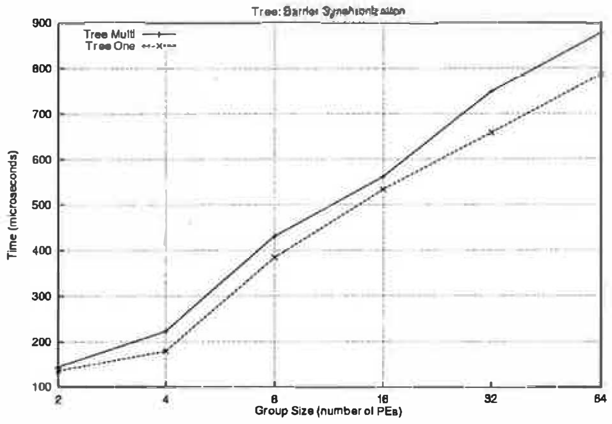
In addition to the theoretical predictions about performance, we carried-out a series of detailed benchmarks on KLAT2's network. Surprisingly, most network benchmarks focus on the performance of communications between a single pair of PCs given that all other processors are not causing network traffic; for a cluster, this is rarely the case. We found the Pallas MPI Benchmarks (PMB) [5] to be among the few benchmarks examining network performance under loading conditions more typical of cluster use. Thus, all the benchmarks we report here come from running PMB on KLAT2.

Aside from the FNN, the only other viable topology supported by KLAT2's inexpensive Fast Ethernet switches is a tree. Fortunately, because KLAT2's FNN includes an uplink switch, we were literally able to benchmark both FNN and tree configurations without physical wiring changes. The FNN benchmarks employed the LAM MPI that we modified to use basic FNN routing. The tree benchmarks forced routing that made the FNN uplink switch behave as the root of a tree; this was easily accomplished by forcing each PC to perform all communications using its NIC 0. Any FNN with an uplink switch can embed a tree network in this manner.

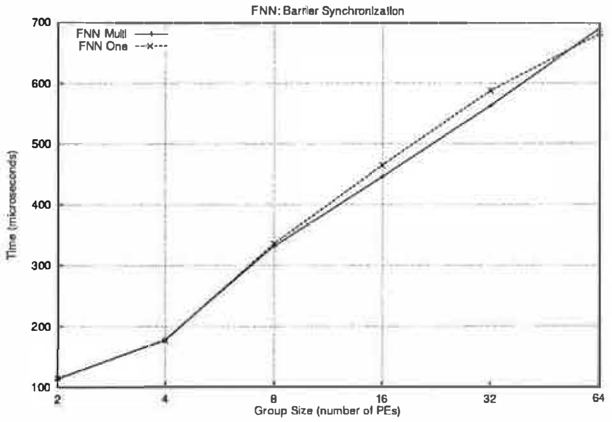
6.2.1. Barrier Synchronization

The first PMB test measured barrier synchronization between processors. A barrier synchronization is an N-way synchronization operation in which no PC is allowed to execute beyond the barrier until all PCs in its group have signaled their arrival at the barrier. KLAT2 has 64 PCs; thus, groups of 2, 4, 8, ... up to 64 PCs could synchronize. For group sizes less than 64, there might be only one (One) group active in the machine at any given moment or there may be multiple (Multi) groups so that all PCs are active simultaneously.

Using KLAT2's network as a tree, one would expect Multi to be slower than One due to interference in the network. The Multi case is slower:



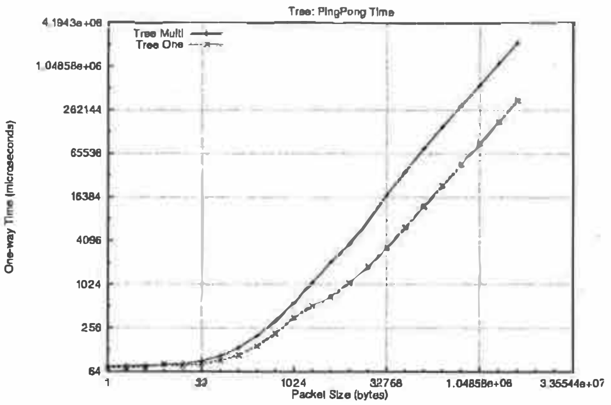
Aside from being generally faster than the tree, the FNN should not suffer a significant penalty for the Multi case. As the following figure shows, there is indeed no penalty for the Multi case; in fact, the Multi case is even slightly faster for groups of size 16 or 32 PCs:



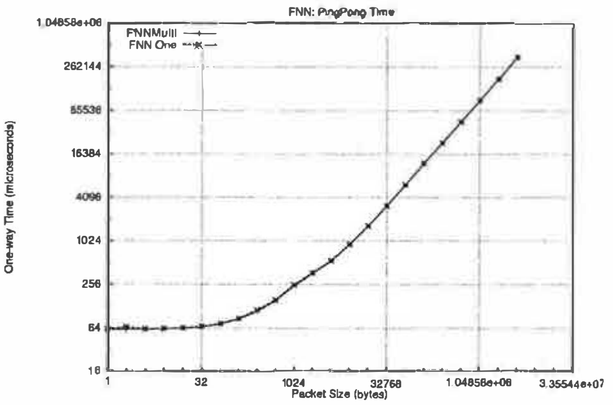
6.2.2. Ping-Pong

The second PMB measurement is a ping-pong test: a message is sent from one machine to another, which sends a message back in response. The time between sending the initial message and receiving the response is twice the end-to-end latency (i.e., twice the one-way time). PMB measures this latency both for only one pair of PCs active (One) and for all PCs active (Multi) in pairs.

Again, we would expect the tree to have poorer performance in the Multi case due to interference, and that is precisely what happens:



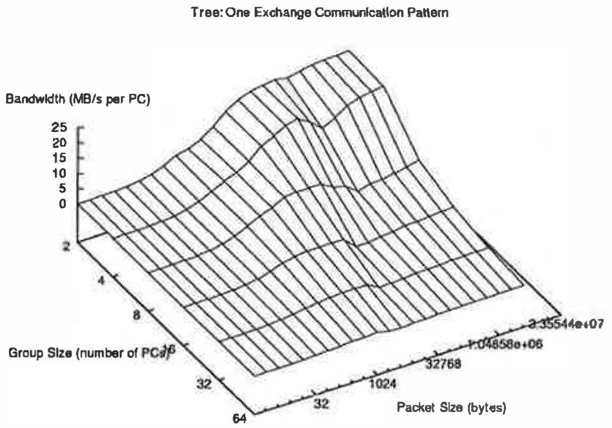
In contrast, the FNN both has lower latency (due to all paths passing through only a single switch) and the One and Multi cases achieve virtually identical performance:



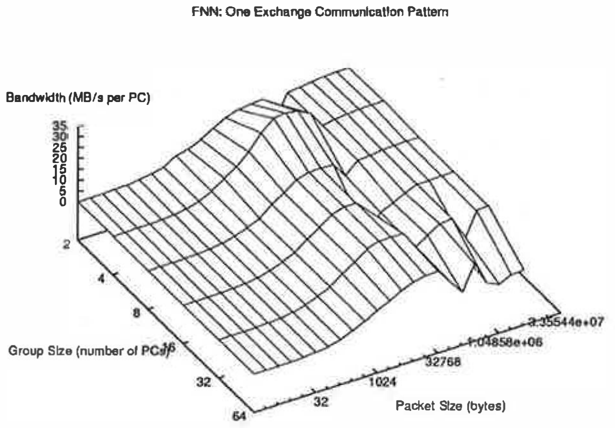
6.2.3. Exchange

The final set of PMB measurements we present involves processors exchanging data simultaneously in both directions along a one-dimensional ring whose length is given as the group size. The One case has only a single group active; Multi has all groups active simultaneously. This type of communication pattern is typical of many grid-structured parallel computations.

For the One case, the tree network actually achieves about 21.5 MB/s — very close to the theoretical peak — in sending 4MB messages between two PCs. However, although some groups may "get lucky" and be connected to the same switch (as the size 2 group happened to be), pairs that span switches severely limit performance of the entire group. Thus, peak performance is excellent, but average performance is relatively poor:

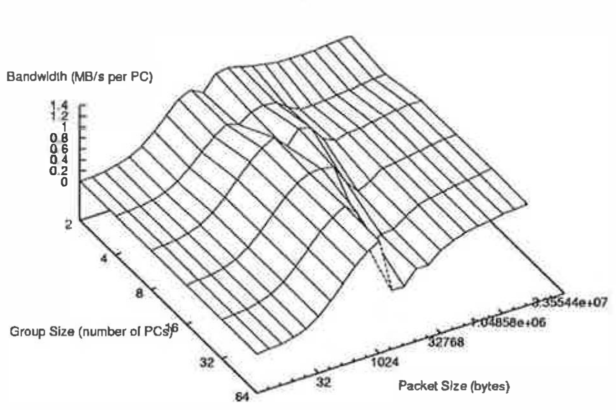


Continuing with the One case, the FNN also achieves very high performance. In fact, because two different NICs can overlap their operation in some cases, bandwidths as high as 31.2 MB/s are achieved. Notice that still higher numbers would be achieved if we were using FNN advanced routing and/or if we had told the GA to optimize for that particular set of communication patterns. Even more significant is the relatively wide and high plateau in the FNN performance; the average case is much better than for the tree. The folds in the graph are real, repeatable, and as yet unexplained, probably the result of an anomaly involving buffer handling:



When we consider the Multi case, we clearly see why a tree network can have such a crippling impact on the performance of many codes. The *best* per-PC bandwidth achieved is only about 1.36MB/s!

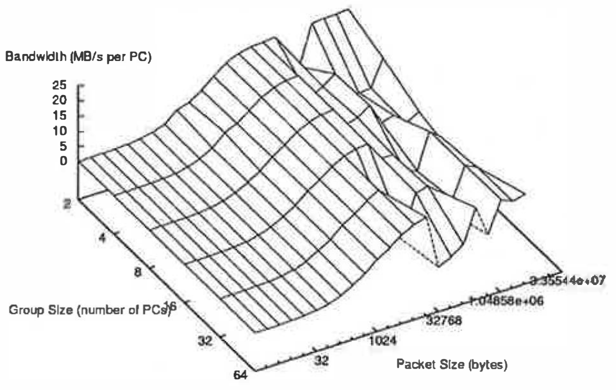
Tree: Multi-Exchange Communication Pattern



This tree performance is especially daunting when one realizes its implications on "channel bonding." Even if we were to duplicate the entire switch tree for 5-way channel bonding, and have the channel bonding be 100% efficient, all of that hardware would yield about 5 times 1.36MB/s, or 6.8MB/s! Worse yet, that is a peak number achieved only for a specific group size and message length... the average performance is even poorer.

In contrast, the Multi case for the FNN yields performance that is only a little lower than it achieves for the One case. The disturbing folds in the graph are even more pronounced, but average performance is well above the peak that could be achieved using channel bonding with trees:

FNN: Multi-Exchange Communication Pattern

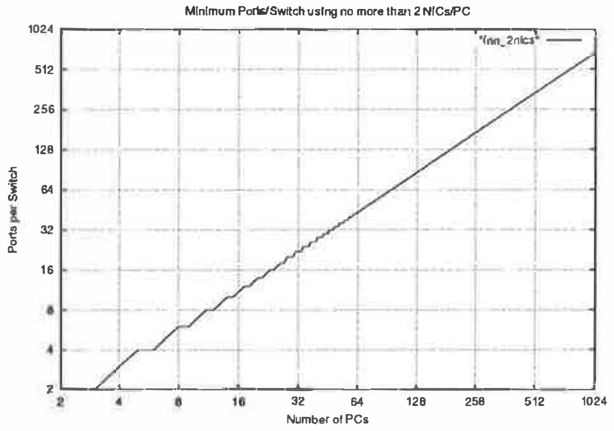


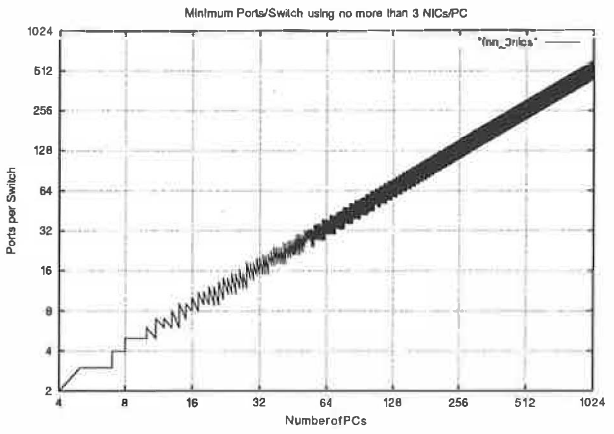
We do not claim to fully understand all the properties of FNNs and we have not yet implemented software that can take full advantage of them. Even at this early point in their development, FNNs clearly offer significant performance — and price — advantages over other network architectures. If all you need is single point-to-point bandwidth, you might not want a FNN; but where bisection bandwidth and/or low latency is the primary concern, as it often is in cluster parallel processing, FNNs are very hard to beat.

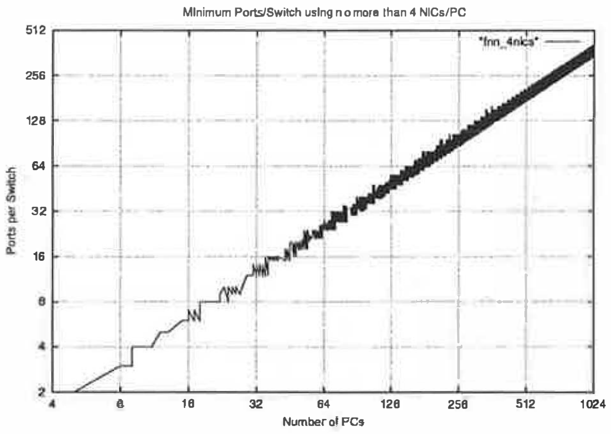
7. Scalability

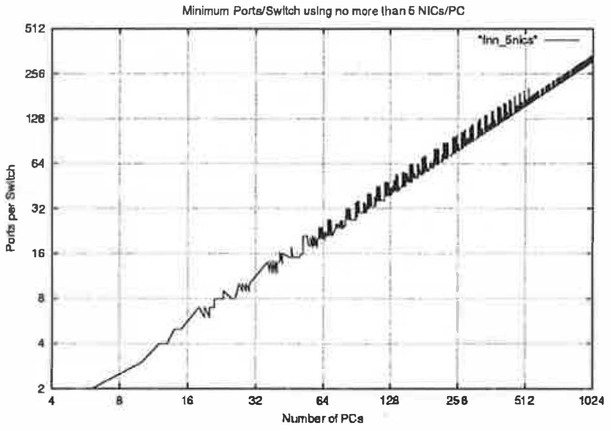
The best way to explore the scalability of FNNs is to directly use the genetic search algorithm for specific system designs; unfortunately, the cost of running the full GA design tool is too high to try a very large number of system sizes. However, for use as an interactive WWW-interfaced CGI program, we constructed a very fast simplified version of the FNN design program that uses simple scaling rules to quickly create reasonably good FNN designs for a given number of PCs, ports per switch, and NICs per PC. Modifying this program made it feasible for us to explore a wide range of system parameters; literally, all system sizes from 3 to 1024 processors.

The following four graphs show how the number of ports per switch must increase to accommodate more PCs in the cluster. Each graph corresponds to a different maximum number of NICs allowed per PC: 2, 3, 4, or 5 NICs/PC. The "ragged" appearance of the curves is due to the fact that, unlike the full GA design tool, this simplified design tool often fails to find the minimum FNN configuration.









Although the depressing reality is that very wide switches are needed to construct a FNN for a very large machine with only a few NICs per PC, this is not as large a problem as it first seems. The ideal is to use switches within a FNN that are capable of full wire speed, but similar benefits can be obtained in very large systems by using moderately-sized switch fabrics (e.g., tree or fat tree of switches) for each switch within the FNN. For example, wire-speed 309-way switches for a FNN connecting 1024 PCs may not be available, but an FNN using 309-way switch fabrics will still yield a strong performance edge over a 1024-way fabric.

8. Conclusions

In this paper, we present a new network architecture: the flat neighborhood network. This single-stage topology makes better use of commodity NICs and switches than traditional topologies, yielding very good latency and outstanding bisection bandwidth with very low cost. It even allows for the network to be engineered, using GA techniques, to optimize performance for specific communication patterns and machine properties. Unfortunately, it also requires a bit of clever restructuring of the usual software interface to the network.

Preliminary results with KLAT2, the first flat neighborhood network machine, show that the expected performance benefits are truly realized. At this writing, we have two primary application codes running on KLAT2. Both of these codes are using our version of LAM MPI modified to use basic FNN routing. The advanced routing is not yet used.

DNSTool is a full CFD (Computational Fluid Dynamics) code, such as normally would be run on a shared-memory machine. Even with KLAT2's FNN, this code spends about 20% of its time in communication. However, it is running on KLAT2 well enough that it is a finalist for a Gordon Bell Price/Performance award [4]. The official price/performance is \$2.75/MFLOPS double-precision and \$1.86/MFLOPS single-precision.

The other application code KLAT2 has been running is the standard LINPACK benchmark. Using ScaLAPACK with our 32-bit floating-point *3DNow!* SWAR extensions, KLAT2 achieves over 65 GFLOPS. The resulting price/performance is better than \$0.64/MFLOPS, making KLAT2 the first general-purpose supercomputer to achieve significantly better than \$1/MFLOPS.

Clearly, neither application could have achieved comparable price/performance without KLAT2's FNN. We believe that FNN architecture can bring similar benefits to a wide range of applications. We anticipate making all the GA network design and routing software fully public domain and have already begun distributing them at: **`http://aggregate.org/`**

9. References

[1] H. G. Dietz and T. I. Mattox, "Compiler Techniques for Flat Neighborhood Networks," *Proceedings of the 13th International workshop on Languages and Compilers for Parallel Computing*, IBM Watson Research Center, Yorktown, New York, August 10-12, 2000, pp. 239-254.

[2] The LAM MPI Implementation, **`http://www.mpi.nd.edu/lam/`**

[3] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, Rice University, Technical Report CRPC-TR94439, April 1994.

[4] Th. Hauser, T.I. Mattox, R.P. LeBeau, H.G. Dietz, P.G. Huang, "High-Cost CFD on a Low-Cost Cluster," Gordon Bell Price/Performance finalist to appear in the *IEEE/ACM Proceedings of SC2000*, Dallas, Texas, November 4-10, 2000.

[5] Pallas MPI Benchmarks (PMB), version 2.2, **`http://www.pallas.com/`**

BLASTH, a BLAS library for dual SMP computer.

Guignon Thomas
Laboratoire ASCI
Orsay, France
`guignon@asci.fr, http://www.asci.fr/`

Abstract

This paper presents a multi-threaded BLAS library for dual SMP Intel computer running Linux. We present simple techniques to obtain parallelism for BLAS call transparently from the client program. We discuss some synchronization methods available under Linux, show performances results for a representative set of BLAS and for a high level linear algebra kernel. We then explain some key points on cache management and how they can impact on performances of the blasth library. Next we'll draw some conclusions on the use of SMP computer for linear algebra and present evolution perspectives for the library.

1 Introduction

BLAS[1][8, 4, 3] are a set of subroutines written in Fortran 77 which provide a standard API for simple linear algebra operations. BLAS are now widely used and base of library such as LAPACK, PETSC, SCALAPACK... A recent development in parallel computing was to use on the shelves components to build high performance computers, these components could include low and dual Intel processor systems. This choice leads to the use of two parallel programming models: message passing between nodes end shared memory inside nodes. The goal of the blasth library is to provide a transparent support of shared memory for BLAS call: the program call BLAS as usual but the call is processed on 2 processors. BLAS are split in 3 level:

- level 1: vector operations,
- level 2: matrix-vector operations,
- level 3: matrix-matrix operations and triangular solve.

In this paper we first present some discussion and experiments for synchronizing threads with Linux on SMP computer, we will then focus on daxpy and ddot level 1 BLAS, dgemm (from level 3). The sequential BLAS library used will be the one from the f77 implementation ASCI Red project[2] and ATLAS[3][9] (for level 3). We also present results for a block LU factorization.

[1]from Basic Linear Algebra Subroutines

2 Base principles

Parallelism in BLAS is transparent from the client program: only a call to setup the execution environment and changing the subroutine calling names is needed. The execution scheme is master-slave: the master process runs the program while the slave process is waiting for instruction for master. When the master issue a BLAS call it tells the slave what job to do by communication via shared memory and each one does his "job part". The master waits for the slave to finish his job and continues the normal execution of the program.

The implementation uses The Linux Thread Library which is included with glibc package. This library provides very simple view of shared memory because each thread has the same memory space (data and stack). Note that The Linux Thread Library does not use real threads but traditional Linux processes sharing their memory space so in the following we will use the 2 words thread/process with the same meaning.

2.1 Synchronization

The master-slave approach relies on process synchronization at start and end of BLAS call. Under Linux process synchronization can usually be done in two way:

- IPC semaphores.
- Thread semaphores.

One third way is to use a shared memory variable (synchronization variable): the slave spins on testing the value change of this variable. In this case the slave is always running even if it doesn't make useful work and a normal (observed) behavior of the Linux scheduler is to place the

[2]http://www.cs.utk.edu/~ghenry/distrib/
[3]http://www.netlib.org/atlas/index.html

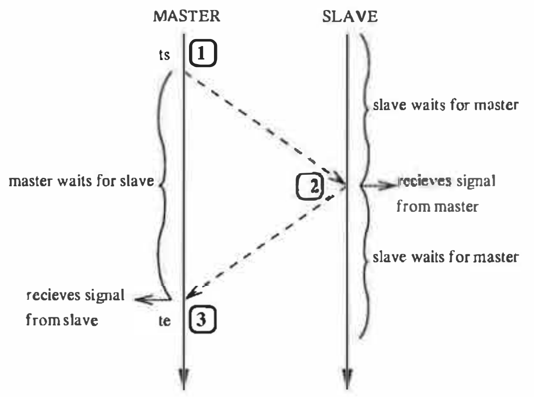


Figure 1: ping-pong test events.

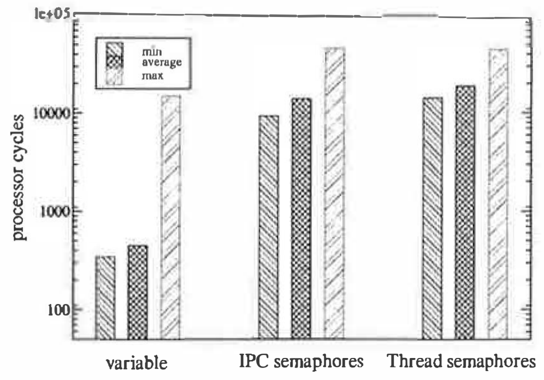


Figure 2: min, average and max times for ping-pong test (1 cycle = 1/400e6 s.).

each running process on different processors so when the master process need synchronization the slave is immediately ready and running.

We compare the three alternatives in a ping-pong test: we make the slave wait for the master and the master "signals" the slave; next the master wait for the slave and the slave "signals" the master . We measure the number of cycles on the master to get the whole job done. Figure 1 may help in understanding the ping-pong test. Moreover each synchronization method must ensure that:

- the slave cross point $\boxed{2}$ if and only if the master cross point $\boxed{1}$,
- the master will go through point $\boxed{3}$ if and only if the slave cross point $\boxed{2}$.

The synchronization variable method fulfills the previous requirements. For IPC and Threads semaphores we must use 2 semaphores[4] each one indicates when the master and slave are ready to enter in parallel section; the ping-pong is done with 2 barriers that act like this:

- each semaphore holds value 0,
- master posts on semaphore 0 (semaphore 0 holds 1) and waits on semaphore 1 value becomes 1.
- slave posts on semaphore 1 (semaphore 1 holds 1) and waits on semaphore 0 value become 1.
- each semaphore holds value 0 again.

Experiments are realized on a dual PII 400 and the time measurement is done using the time stamp counter[5] (tsc) of Intel Pentium processors, we suppose that the tsc of each processor holds roughly the same value. Results are presented figure 2: for each method we make 1000 runs and presents the smallest, average and largest time for ping-pong. These results show that synchronization variable is by far the fastest method. As we said previously the difference between synchronization variable and the 2 other methods is that the slave process is ready and running so synchronization does not pay the cost of a system call and moving process from the wait/suspend queue to the running queue. On the other hand having an active slave may interfere with other multi-threaded library.

[4]it can also be done with mutexes.

[5]this is a 64 bits counter which value is incremented each cycle and start counting at power on of processor

2.2 Passing parameters and data sharing

Passing BLAS parameters to the slave is realized by writing in a shared variable the address of the first parameter of the BLAS call before synchronization as shown in the following example `env_base` holds a pointer to the data needed by slave process and `env_blasth_signal_value` holds the function to be run by slave:

```
void **env_base;
void (*env_blasth_signal_value)();

void blasth_daxpy(const int *n,
                  double *alpha,
                  double *X,
                  const int *incx,
                  double *Y,
                  const int *incy){
 // realize Y = *alpha * X + Y
 // where X and Y are vectors of
 // size *n with respective increments
 // of *incx and *incy
 // executed by the
 // master from the
 // application program

 env_base = (void **)&n;
 env_blasth_signal_value = TH_DAXPY;

 // tell the slave there is
 // some job to do
 blasth_master_sync();
 // some job

 // wait for the slave
 blasth_master_sync_end();
}


void blasth(){
 // excuted by the slave from the
 // environement setup

 while(1){
  // wait for the master
  blasth_sync();

  //call the function set by the master
  env_blasth_signal_value();
 }
}

TH_DAXPY(){
 // at this point env_base
 // contains a pointer to the
 // first needed parameter
 // (int *)env_base[0] is a
 // pointer to the size of vectors (*n)
 // (double *)env_base[1] is a
 // pointer to the scaling factor (*alpha)
 // (double *)env_base[2] is a
 // pointer to the first element
 // of vector X
 // ....

 // some job

 // tell the master
 // that job is finished
 blasth_sync_end();
}
```

The blasth_daxpy calling sequence is identical to the daxpy calling sequence from a C program (the BLAS library is originally written in f77 so the API is f77 compliant) and the parameters are written before the synchronization variable so the strong memory ordering (for write operations) of the Pentium processor family ensure that slave process will see exactly the same parameters in TH_DAXPY as the master in blasth_daxpy.

Data sharing is done by splitting the result between the master and the slave: if the result is a vector the master has to construct the first half and slave has to construct the second half; if the result is a matrix of size $m \times n$ the master will construct either the first $n/2$ columns or $m/2$ rows and the slave will construct the remaining columns or rows. We show splitting examples in figure 3 for dgemv and dgemm (respectively matrix vector product and matrix matrix product).

We does not use cycling split of data to avoid cache line

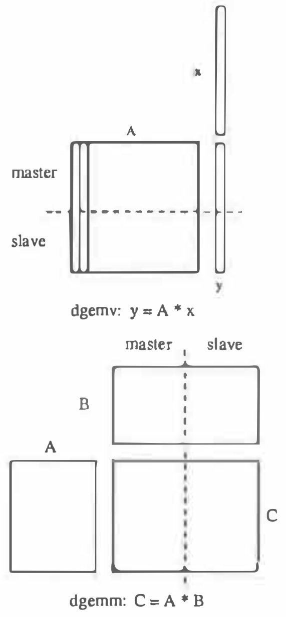


Figure 3: data splitting for dgemv and dgemm

sharing between processors (especially when writing data). The splitting are also chosen to avoid temporary data which would require dynamic allocation.

3 Some results

The results presented here where done on a dual PII 400 system running Linux The time measurements are done using the time stamp counter of the processors. The base BLAS libraries used are from the Fortran 77 implementation[6], ASCI Red project and ATLAS project (for level 3). The memory bandwidth measurements are done with the hardware performances counter available on the Pentium Pro family processors[7]. For each BLAS we test we compare single and dual performances in two case, with cache memory flushed out (datas are read from main memory) and with cache memory loaded: the test is done several times before measuring execution time (remark: this does not seem that data will fit into cache).

3.1 daxpy and ddot

daxpy and ddot are the two main level 1 BLAS, daxpy is a linear combination of vectors $y = \alpha.x + y$ and ddot is a dot product of the form $\alpha = x^T.y$ (the "d" before each name indicates that the BLAS use double precision floating point arithmetics).

[6]with recent version f77 level 1 blas compete with those of ASCI Red project

[7]an introduction on how to use these counters is available at http://www.cs.utk.edu/ ghenry/distrib/mon_counters

Performances results for daxpy are presented figure 4 and acceleration is showed 5. Observed results are typical for level 1 operations: we observe a peak when data fits into L1 cache and a smooth decrease when data does not fits in L1 but remains in L2. Performance from main memory is driven by memory bandwidth. Acceleration is quite good for large vectors especially when datas are in cache.

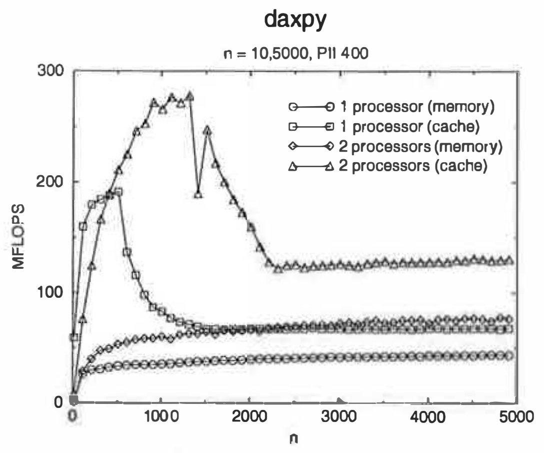


Figure 4: daxpy performances with 1 and 2 processors

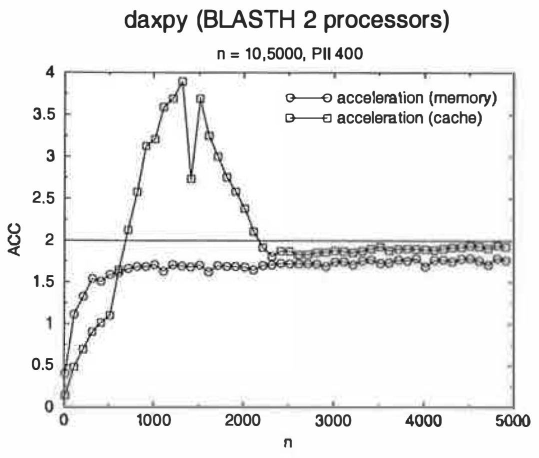


Figure 5: daxpy acceleration for 2 processors

Performance results for ddot are showed figure 6 and figure 7 for acceleration. Comments on performances results are the same as for daxpy but we observed that scaling is not as good as for daxpy: cache operation scaling is roughly 1.8 but remains good. On the other side memory operation scaling in poor (≤ 1.5). To understand that we make some memory bandwidth measurements with very large vectors to not consider time spent in synchronization and the results are presented in table 1: we see that single processor ddot use more than half of the theoretical peak memory bandwidth (the system uses pc100 SDRAM allowing 800e6 B/s memory bandwidth) and dual processor uses up to 75% of available bandwidth which seems very good for the test system.

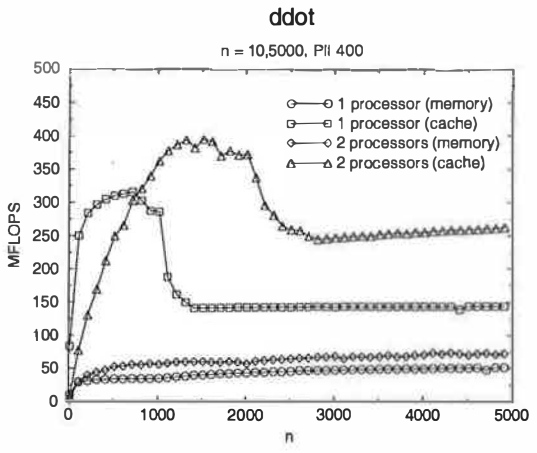


Figure 6: ddot performances with 1 and 2 processors.

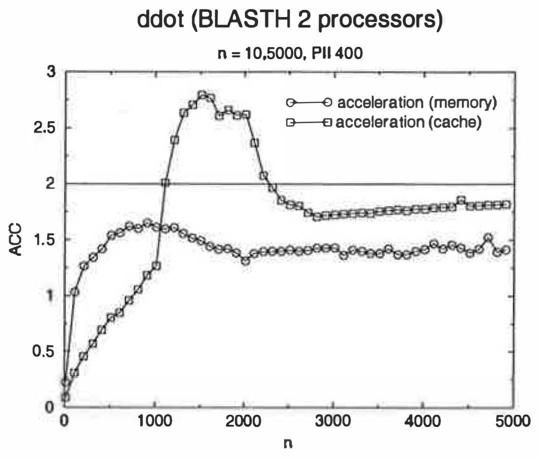


Figure 7: ddot acceleration for 2 processors.

The daxpy and ddot BLAS show good acceleration with the blasth library when datas are present into cache memory, this behavior is in fact common to all level 1 operations since each component of vectors is used only one time in computation (no temporal locality). Operations on small vectors ($n \leq 1000$) will not scale due synchronization cost compared to the small number of floating point operations issued (n or $2.n$ for level 1 BLAS). The memory bandwidth is the key point for scalability when datas are out of cache which is always true for very large data sets.

3.2 dgemm and block LU factorization

dgemm is a level 3 operation which performs a matrix matrix product $C = \alpha.A.B + \beta.C$ where A, B, C are respec-

nproc	n	memory bandwidth (1e6 B/s)
1	10000	440
1	100000	470
2	10000	590
2	100000	600

Table 1: memory bandwidth used by ddot with 1 and 2 processors.

tively $m \times k$, $k \times n$ and $m \times n$ matrices. The ASCI Red implementation and ATLAS use a block method to perform matrix matrix multiply and achieve good performances As we see in figure 8 end figure 9 performances and acceleration are very good and remain as the size of matrices increase. Performances for dgemm are important because it's a building block for block LU factorization. We know outline a block LU factorization method, the reader may refer to [5] for an in-deep analysis and algorithms, and discuss how multi-threaded version can be realized.

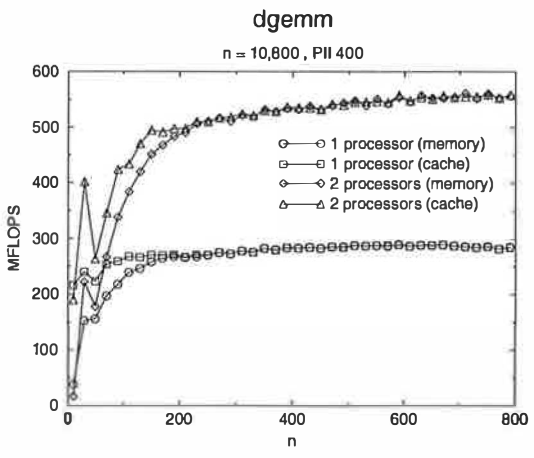


Figure 8: dgemm performances with 1 and 2 processors

LU factorization is used to solve linear system like $A.X = B$ by factorize A into $L.U$ where L is a lower triangular unit matrix and U is an upper triangular matrix. Efficient LU methods rely on matrix matrix product and we outline such method in the following. For simplicity we suppose that the block size n_b divide n which is the size of the square matrix A and set $N = n/n_b$. Each block of A is numbered $A_{i,j}$ $i, j = 1..N$. The block LU performs as follow:

1. set $k = 1$,
2. compute $A_{k,k} = L.U$ and overwrite $A_{k,k}$ with L and U using a non blocked LU factorization,
3. apply row interchange in $A_{k,k+1:N}$ and in $A_{k,1:k-1}$
4. solve $L.X = A_{k,k+1:N}$ and overwrite $A_{k,k+1:N}$ with the solution,

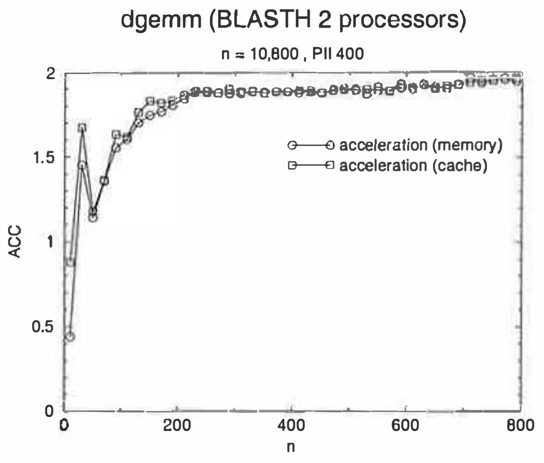


Figure 9: dgemm acceleration for 2 processors

5. solve $X.U = A_{k+1:N,k}$ and overwrite $A_{k+1:N,k}$ with the solution,
6. update $A_{k+1:N,k+1:N}$ with $A_{k+1:N,k+1:N} = A_{k+1:N,k+1:N} - A_{k+1:N,k}.A_{k,k+1:N}$,
7. if $k < N$ set $k = k + 1$ and go to step 2.

Step 2 uses subroutine dgetf2 from LAPACK, row interchange is done using dlaswp, step 3 and 4 use dtrsm (triangular solve with multiple right hand side) and step 5 uses dgemm. It is important to note that most computation is done in steps 4 and 5 which use level 3 BLAS and represent $1 - 1/N^2$ of the floating point operations issued (see[5]). Figure 10 presents task dependencies in block LU for step k, it outlines a graph dependency for a parallel implementation. At this point we have two choices for multi-threaded version:

- We can use a dependency graph of task with each processors searching for ready tasks (tasks for which each ancestor is done) and executing them. By splitting the solve and matrix multiply tasks into smaller ones we can see that LU for step k can be done while matrix multiply for step $k - 1$ is finishing since $A_{k,k} = A_{k,k} - A_{k,k-1}.A_{k-1,k}$ is done at step $k - 1$.
- We can also perform the LU task on one processor and then use multi-threaded version of dlaswp, dtrsm and dgemm.

The first solution requires to construct the dependency graph and more complex synchronization scheme but always based on synchronization variable. Construct the graph and searching for ready task may be a serious overhead thus limiting acceleration. The second solution let the LU task be a sequential bottleneck and use parallelism only on level 3 operations.

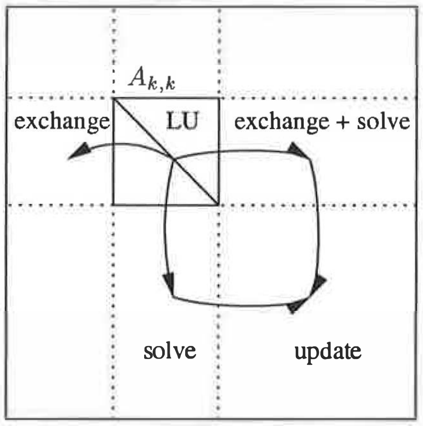


Figure 10: LU factorization scheme for the step k

For implementation we choose the second solution because it's simple and LU bottleneck is in fact very small since it represents only $1/N^2$ of the overall job. Results for our block LU factorization are presented figure 11 and figure 12: we use dgetrf from ATLAS as a reference code for our sequential block LU. Acceleration figure presents two curves; one presents acceleration obtained by the multi-threaded block LU, the other (ideal acceleration) is computed using the sequential code by looking for time spent in dgetf2 and in level 3 operation:

$$acc_{ideal} = \frac{1}{t_{dgetf2} + t_{level3}/2}$$

For $n \geq 1000$ acceleration is relatively closed to ideal acceleration but smaller case acceleration is far from ideal due to synchronization cost.

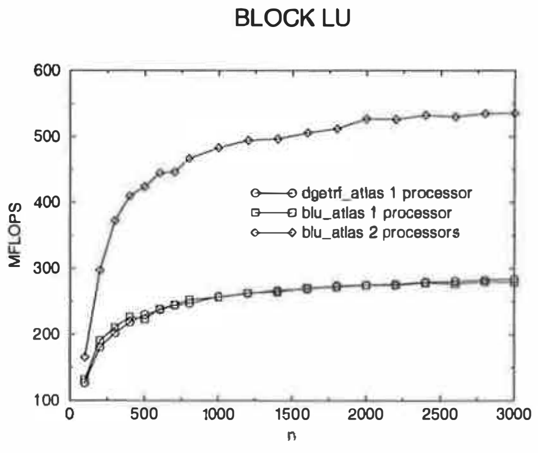


Figure 11: block LU performances with 1 and 2 processors

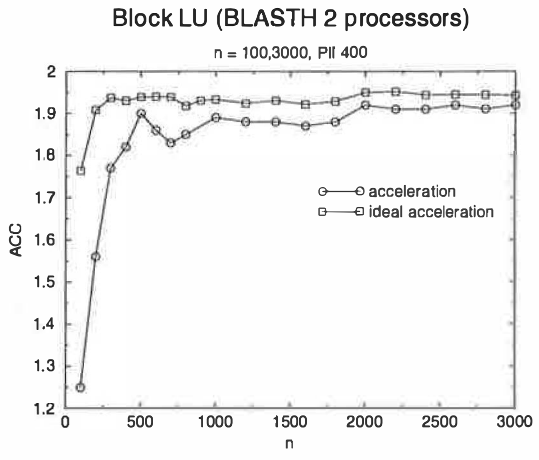


Figure 12: block LU acceleration with 1 and 2 processors

4 Memory bandwidth and cache use

In this section we present some well known key points of cache use and how they impact on performances in BLAS subroutines for single and dual cpu execution, we will discuss on effect of non continuous data, false sharing, mutu al exclude, data blocking, stack alignment and thread/processor affinity. All examples suppose we are using an Intel P6 class processor which suppose that L1 caches lines are 32 bytes long and L1 is 2 way set associative, reader may refer to[1] for full information on optimizing codes for Pentium processors.

continuous datas.

Level 1 BLAS use loops that access arrays in a sequential manner; if we suppose that an array t[n] of double is cache line aligned (for simplicity) accessing t[0] loads t[0],t[1],t[2],t[3] into one level 1 cache line, then following array cell accesses may not use memory until we access t[4]. This situation makes the ration of useful loads[8] on effective loads[9] be 1. By using vector increments of 2, only even cells are used which make t real size becomes 2n, and the previous ration becomes 0.5. This is a first argument to avoid cycling split of vectors for multi-threaded level 1 BLAS because master and slave processes will use more memory access to do the same job.

false sharing.

The Intel P6 family use a cache coherency protocol with four states usually called MESI[10]. This protocol is write invalidate which means that when two or more processors holds a copy of the same memory line if one of them writes

[8]loads necessary to perform computation
[9]loads effectively issued
[10]Modified, Exclusive, Shared and Invalid

in, the cache line holding the memory line is invalidated on other processors. False sharing occur when processors write to a shared cache line but not at the same location: there is no real coherency problem since processors write to different location and since the cache allocate a line when a write misses[11] the protocol makes each processor invalidate the other forcing reload of a cache line at each write. This situation occurs with blasth library at the boundary of results blocks but in the case of level 1 BLAS only one cache line will be shared between 2 processors. In the case of dgemm for a $m \times n$ matrix up to $max(m, n)$ caches lines can be shared but usual optimization of dgemm use block copy of the resulting matrix avoiding such situation. False sharing is another argument to avoid cycling split and we can see effect on daxpy in figure 13: there is no cache effect on operand y while operand x is accessed by each thread with an increment of 2.

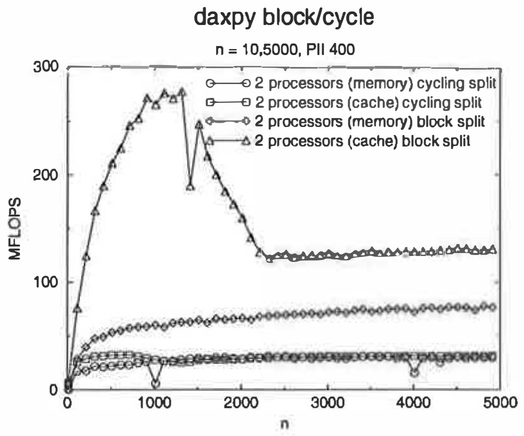


Figure 13: Effects of false sharing with daxpy on 2 processors

mutual exclusion.

Mutual exclusion appends when 2 or more memory lines are needed but cannot be in cache all-together because they fit in the same cache line and successive access cause exclusion of the memory lines previously loaded. This is a real problem for direct mapped caches where a memory line can be in only one cache line. N way set associative caches solve this problem by allowing a memory line to be in N different cache lines, the N locations are called a set; this is the case of the P6 processors where L1 cache is 2 way set associative and it avoids mutual exclusion for all Level 1 BLAS with less than 3 vector operands (like ddot and daxpy). Mutual exclusion can also appends for matrix operations like dgemm: when using a block method, leading dimension[12] can be such that some memory lines share the same set into cache avoiding more than N of them at the same time: on figure 14 the memory lines l1 l2 and l3 are 1024 doubles spaced (remember that PII L1 cache has 256 sets each containing 2 line of 32 bytes) so they fit into the same L1 cache set which can hold only 2 different memory lines. Thus l1, l2 and l3 and subsequent lines exclude mutually. The solution to have the block in cache is to make a copy into contiguous memory and this is the solution adopted in ATLAS.

[11]PII and up use a write-back/write-allocate strategy while the PPRO use write-trough.

[12]distance between first elements of columns.

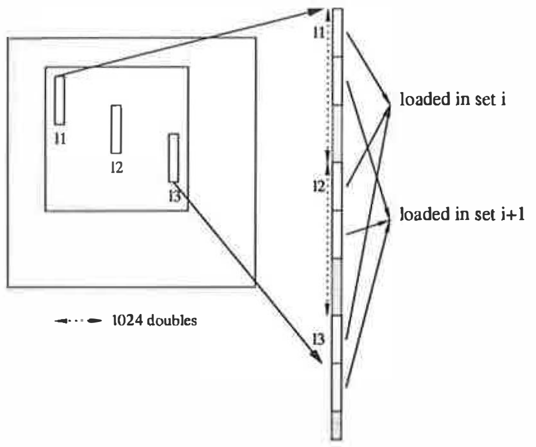


Figure 14: Mutual exclusion in block access.

data blocking.

Blocking is an optimization technique that allows a full cache use and thus reduces memory bandwidth usage. Blocking is no always possible: for BLAS 1 and 2 the majority of data is accessed only one time making temporal locality very low; on the other hand Level 3 operations use a three nested loop structure with 2 dimension matrices making each matrix elements accessed more than on time. We give an example for matrix matrix product $C = A.B$ where C, A and B are respectively $m \times n$, $m \times p$ and $p \times n$ matrices; in this case each element of A and B are accessed respectively n and m times. The block method for matrix matrix product generally consist of:

- split result matrix C into blocks $C_{i,j}$ of size $n_b \times n_b$, each blocks is constructed into a continuous array C_b which is then copied back into the right $C_{i,j}$.
- matrices A and B are split into panels A_i and B_j of size $n_b \times m$ and $k \times n_b$ each panel is copied into continuous arrays A_b and B_b. The choice of n_b must ensure that C_b A_b and B_b fit into one level of cache, usually L2 cache.

then:

```
1: for $i = 1$ to $m/n_b$ do
2:     $A_b \leftarrow A_i$
```

```
3:   for $j = 1$ to $n/n_b$ do
4:      $B_b \leftarrow B_j$
5:      $C_b \leftarrow 0,$
6:      for $k = 1$ to $p/n_b$ do
7:         $C_b \leftarrow C_b + A_{bk}.B_{bk},$
8:      end for
9:      $C_{i,j} \leftarrow C_b$
10:  end for
11: end for
```

We suppose for simplicity that n_b divides m, n and p. Figure 15 may help in understanding operations performed on blocks. In the case of the previous algorithm matrix A is loaded only one time into cache compared to the n times access of a classical ijk loop while matrix B is still accessed m times. This simple block method greatly reduce memory access and real codes may choose by looking at matrix size which loop structure (ijk vs. jik) is best appropriate and if some matrix operand fits totally into cache.

In the previous we where working on L2 cache and we does no talk about L1 cache use. In fact L1 will be generally too small to handle a $C_{i,j}$ block and one panel of A and B but remember that operation performed at step 7 of the previous algorithm is a matrix matrix product so each operand A_{bk} and B_{bk} is accessed n_b times: this part could also use a block method. Since n_b is relatively small the implementation may load only one of C_b, A_{bk}, B_{bk} into L1 cache and works with others from L2 cache. The reader may refer to ATLAS source code and description for a complete analysis and test of block methods in various environments. Another projects for fast matrix matrix multiply are Phipac[13][6] and the BLAIS[7] library from MTL[14].

stack alignment and thread/processor affinity.

Stack alignment has been an issue because older gcc versions cause doubles not being aligned on an 8 bytes boundary which make access cost extra cycles. We have face this problem with level 3 BLAS that use fixed size arrays on stack for blocking resulting in poor performances. Recent gcc versions (i.e. 2.95.x) solve this problem and propose various option to control the stack alignment such as -malign-double and -mpreferred-stack-boundary=x.

Thread/processor affinity is a general issue in smp systems; the cache efficiency can be reduce if the task scheduler moves thread from one processor to another. At this time there is no way to force thread/processor affinity on a standard Linux kernel but as we said in section 2.1 the normal behavior of the Linux scheduler is to place each running process (master and slave) on 2 different processors and a kernel patch is available at http://isunix.it.ilstu.edu/~thockin/pset/ that add some control on the thread/processor binding.

[13]http://www.icsi.berkeley.edu/~bilmes/phipac/
[14]http://www.lsc.nd.edu/research/mtl/

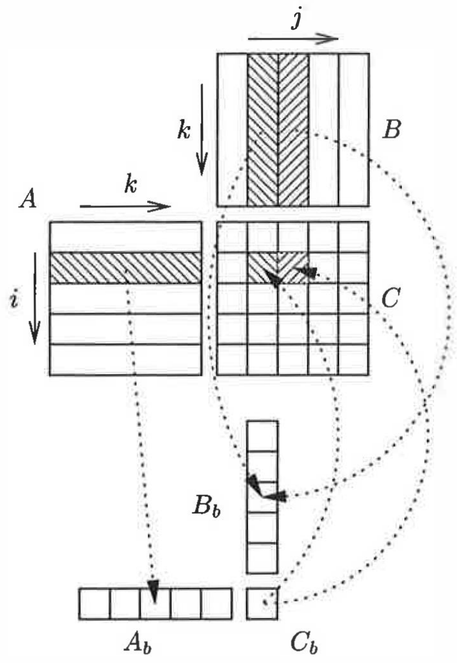


Figure 15: Block matrix matrix product.

5 Conclusions and perspectives

The use of low end Intel SMP computers inside Beowulf or c.o.w. can help in getting better performances when applications does not consume a lot of memory bandwidth: we have always to remember that a cluster of single processors nodes has twice aggregate memory bandwidth of an equivalent cluster of dual SMP with the same number of processors. In linear Algebra a lots of high level computation kernel use block methods and the blasth library could help in these cases as we see for LU factorization. The perspectives for this work are important because we need to work on more LAPACK routines to provide a useful library. There is also work to do on matrix matrix product and LU factorization to improve acceleration in small cases and our choice of simple parallelization for LU cannot be ideal for more processors. Porting to other platforms such as alpha systems is an ongoing work and theses systems can change scalability results for high bandwidth consuming BLAS such as ddot and dgemv making the blasth library more interesting.

References

[1] Intel Architecture Optimization Manual. Order Number 242816-003, 1997.

[2] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. Corrigenda: "An extended set of FORTRAN Basic Linear Algebra Subpro-

grams". *ACM Transactions on Mathematical Software*, 14(4):399–399, December 1988. See [4].

[3] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, March 1990.

[4] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, March 1988. See also [2].

[5] Ch. F. Van Loan G. H. Golub. *Matrix computations*. The Johns Hopkins University Press, third edition, 1996.

[6] C.W. Chin J. Bilmes, K. Asanovic and J. Demmel. The PHiPAC matrix-multiply distribution. Technical Report TR-98-35, International Computer Science Institute, Brekeley CA, 94704, October 1998.

[7] Andrew Lumsdaine Jeremy G. Siek. A rational approach to portable high performance: The Basic Linear Algebra Instruction Set (BLAIS) and the Fixed Algorithm Size Template (FAST) Library. In *2th European Conference on Object-Oriented Programming, workshop on Parallel Object-Oriented Scientific Computing (POOSC'98)*, Brussels, Belgium, july 1998.

[8] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Software*, 5(3):308–323, 1979.

[9] Jack J. Dongarra R. Clint Whaley. Automatically Tuned Linear Algebra Software. In *SuperComputing '98 Proceedings*, 1998.

Sequence Analysis on a 216-Processor Beowulf Cluster

Katerina Michalickova
Dept. of Biochemistry, University of Toronto and Samuel Lunenfeld Research Institute, Toronto, ON, Canada
katerina@mshri.on.ca

Moyez Dharsee
Samuel Lunenfeld Research Institute, Toronto, ON, Canada
dharsee@mshri.on.ca

Christopher W. V. Hogue
Dept. of Biochemistry, University of Toronto and Samuel Lunenfeld Research Institute, Toronto, ON, Canada
hogue@mshri.on.ca

Abstract

In this work we describe the implementation of a 216-processor Beowulf cluster with switched gigabit Ethernet networking. This design includes the use of a 8-CPU high performance midrange computer with 8 gigabit ports as a cluster head, a design that limits I/O contention. We have been developing applications software for bioinformatics research in protein folding, as well as the MoBiDiCK system for managing cluster applications that is extensible to general purpose distributed computing. In addition to the cluster architecture, we present a new cluster application for bioinformatics, a variant of the BLAST family of sequence comparison programs. MOBLAST performs the BLAST algorithm in an exhaustive manner, avoiding its initial heuristic approach to finding hits. This effectively slows BLAST down to approach the speed of other comprehensive search methods such as a Smith-Waterman alignment. MOBLAST requires a sizeable cluster to run. We describe the development of MOBLAST and its use in making an exhaustive M×N database of alignments where M is the set of protein sequences with known 3-D structures, and N is the set of all protein sequences. This M×N database of protein alignments will facilitate further research in protein folding, the ultimate aim of our work with Beowulf cluster technology. Furthermore, we describe a general algorithm for partitioning M×N problems and implement this in the MoBiDiCK computing model.

1. Introduction

1.1. Bioinformatics

We define bioinformatics as the science of testing biological hypotheses using informatics approaches. High-throughput biology has been producing a wealth of new information in the form of DNA and protein sequences, 3-dimensional molecular structures as well as newer data relating to the interconnected networks of molecular interactions that form the molecular machines of life. We are interested in a fundamental problem, the protein folding problem. We have been early adopters of Linux and Beowulf clusters in adapting these computational resources through the creation of new software in order to tackle the compute-intensive problems in bioinformatics.

1.1. M×N sequence comparisons – assigning local structure to sequence

Most of the research in bioinformatics focuses on functional assignment of proteins, the working molecules forming the machinery inside a cell. So far the best results of assigning function to protein sequences come from studies of sequence similarity. A routine operation for a biologist is to query a new protein sequence against a sequence database to obtain a list of similar sequences, then use these results to infer the function of the query sequence. The core algorithms for comparing sequences involve making alignments of sequences and finding regions of sequence that are similar according to a statistical model. Algorithms that find optimal alignments can be time-consuming when run over a large database. The BLAST[1] family of programs are widely used for searching DNA and protein databases for sequence similarities; they employ a heuristic for speeding up the search[2]. Unlike a Smith-Waterman alignment[3] BLAST does not perform an exhaustive optimization of the query against every sequence in the database. Instead it scans the database with a heuristic and only when the condition of the BLAST heuristic is satisfied is the optimal alignment computed, scored and reported.

We are interested in queries of protein sequences with known 3-dimensional structures. With the addition of

3-D information, even a very short alignment can provide significant information to a biologist[4], as it may be inferred that the local 3-D structures of similar sequences are also found in the query sequence. Short alignments are often skipped by the heuristics in BLAST. We have decided to build a cluster-based BLAST application that bypasses the heuristic and performs the exhaustive alignment between the query and each sequence in the database using the core BLAST alignment routines. We present MOBLAST, a cluster application integrated with the MoBiDiCK[5] distributed computing system, developed using the NCBI software development toolkit[6]. MOBLAST performs exhaustive protein-protein BLAST alignments in an M×N manner and reports BLAST statistics and alignments.

To obtain an exhaustive comparison of each protein with 3-D information against the database of known protein sequences, we have implemented software to compare these to a non-redundant protein sequence database (NR) in an M×N fashion. In this case N is the protein sequence database size, and M is the set of sequences from known 3-D structures. We are also aware that other biological comparisons are useful to perform in an M×N fashion[7]. This is because it is often faster to retrieve a precomputed comparison from a database than to compute the result on the fly. Databases of precomputed comparisons are valuable resources for biologists, especially in clustering similar sequences or 3-D structures[7][8]. We report a general method to parameterize an M×N comparison using the MoBiDiCK system that avoids processing symmetric duplicate comparisons in the upper triangle of an overlapping range in a database.

2. Cluster architecture

Our cluster consists of 108 independent 2U rack-mount dual-PIII 450 MHz CPU computers, which contribute a total of 216 CPUs to the system as a Beowulf cluster. These nodes are interconnected with fiber optic cable using Gigabit Ethernet with two 64-port Gigabit Ethernet switches. In addition, the cluster has a "head" computer, a high performance HP N-Class server with 8 64-bit PA-RISC CPUs. The "head" initiates compute jobs and collects the output from the nodes, and it stores results on a high-performance Fiber-Channel RAID system. The head has 8 Gigabit Ethernet ports connecting through 4 fibers to each of the two switches.

The headed cluster was chosen as an architecture because of some severe I/O contention we noticed on our smaller 32-processor cluster as jobs completed on nodes simultaneously with our protein folding software applications, which are described elsewhere[8]. This new architecture enables a large number of simultaneous data exchanges with many nodes in the cluster. This is possible because the head computer has approximately 14 times the bandwidth of a single node, owing to the 8 64-bit PCI Gigabit cards in the head, versus 1 32-bit PCI Gigabit card in each node.

Although the architecture we have selected is certainly not the least expensive route, we selected components and vendors by focusing on the ability to upgrade the cluster in a cost-effective manner. We estimate that doubling the current system's performance will cost ¼ of the initial expenditure. We have identified the following options to upgrade the current architecture.

Upgrade Paths – Nodes. We selected rackmount nodes that employ standard components, and rejected those we considered overly customized. The cases we have selected together with the power supplies and the hard disks can all be conserved on upgrade. Each of the 2U rackmount computers can be upgraded with industry standard ATX or NTX motherboards, RAM and new CPUs. Future upgrades could be selected from Intel, Alpha or even PowerPC architectures, whichever is most cost effective.

Upgrade Paths – Head. The HP N-class was selected based on knowledge that it is one of the latest designs on the market. The head computer can be field-upgraded to faster processors, more RAM and more high speed disk. It is CPU-upgradable to the Intel/HP 64-bit EPIC architecture chips, commonly known as "Merced", (released with the name "Itanium"). Indications are that this will be capable of running 64-bit Linux, however upgrades to faster versions of the PA-RISC architecture CPU and HP-UX are also planned by HP. The "head" is also capable of being upgraded to 64 Gb of RAM.

Upgrade Paths – Networking. The existing network comprising 64-bit PCI network cards, fiber optic cabling, and switches can be conserved. The 64-bit PCI network cards are currently only being used with a 32-bit PCI bus; their speed is not fully realized in the current nodes. We recently learned that Foundry Networks will begin shipping a chassis that is twice as large and supports the same "blades" in our current switch. We can upgrade by consolidating our existing Gigabit "blades" into a single 128-port wire-speed crossbar switch as the cluster backplane. This can allow us to add 10 more nodes from the ports freed up in consolidating the two switches, or perhaps contemplate doubling the size of the cluster.

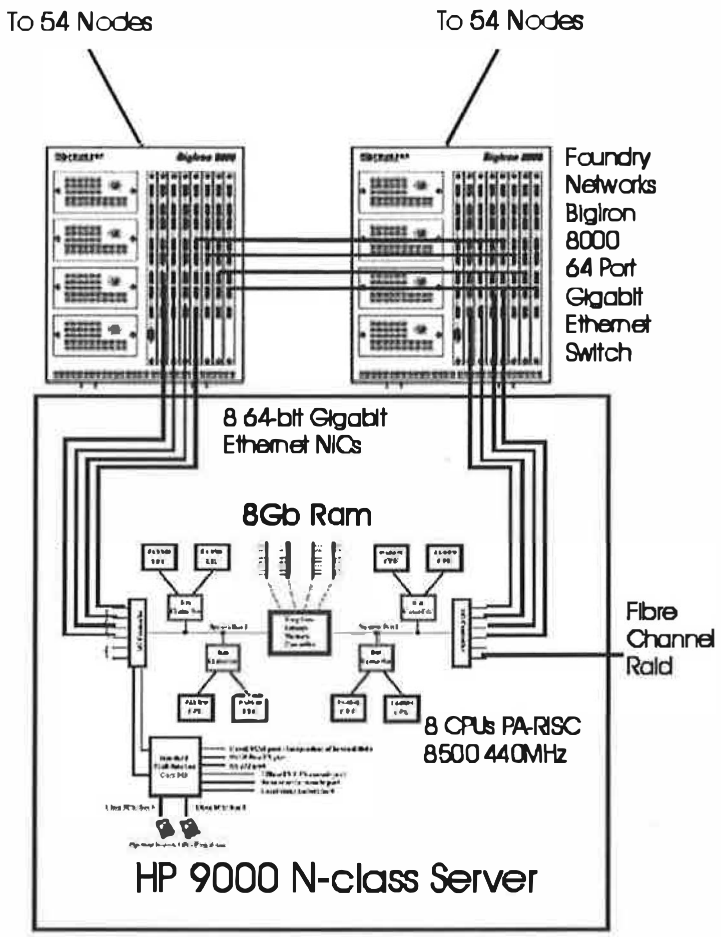

To 54 Nodes
To 54 Nodes
Foundry Networks BigIron 8000 64 Port Gigabit Ethernet Switch
8 64-bit Gigabit Ethernet NICs
8Gb Ram
Fibre Channel Raid
8 CPUs PA-RISC 8500 440MHz
HP 9000 N-class Server

Figure 1. YAC Networking and Head Architecture. In addition to 54 fiber optic cables connecting nodes, each switch has 2 up-links in a ring topology with additional switches on our LAN, accounting for all 128 ports. Photographs of the cluster are at http://bioinfo.mshri.on.ca/yac/.

Table 1. Cluster Specifications

Overall:	
RAM:	64 Gb
HD:	1.6 Tb Disk Storage
CPUs:	224
NIC:	Gigabit Ethernet
OS:	Linux/HP-Unix
PRICE:	$1,300,000 CDN ($884,000 US)
Cluster Head	
Manufacturer:	Hewlett Packard
Model:	9000 N-class Server
RAM:	8Gb
HD:	300 Gb AutoRAID
CPU:	8 440 MHz PA-RISC 8500 64bit
NIC:	8 Gigabit Ethernet, 64 bit PCI cards
OS:	HP-UX 11.0
108 Cluster Nodes	
Manufacturer:	VA Linux (94 nodes + 14 from original cluster)
Model:	FullOn 2x2
RAM:	512K
HD:	14Gb IBM
CPU:	2 450 MHz Intel Pentium III
NIC:	1 Intel Pro/1000 Gigabit Ethernet, 1 integrated Intel 100/BT
OS:	Linux (Custom VALinux SMP kernel)
2 Network Switches	
Manufacturer:	Foundry Networks
Model:	BigIron 8000
NIC:	64 Gigabit Ethernet Ports
Air Conditioner	
Manufacturer:	Carrier
Model:	40RM-014-B600HC
Capacity:	12.5 Ton with Outdoor Roof-Mounted Chiller
6 Racks	
Manufacturer:	DL Custom
Model:	Controller Relay Rack

3. Administration

While the cluster is fully capable of running MPI or PVM based parallel software, we have focused on developing cluster middleware that allows us to perform computations with a minimum of administrative effort. This is important to us for three reasons. First, we are limited in that we do not have operating budgets required to administer a complex system. Second, we need to make our cluster based software accessible to Biologists who may be accustomed to a Web-based interface for running software. Last, we wish to be able to branch out from cluster computing to wider-area distributed computing, and we have sought to develop a mechanism by which we could make use of other computers and clusters over the Internet with MoBiDiCK distributed computing system.

Using MoBiDiCK, a computational task can be partitioned into subtasks and distributed over a set of Web server nodes. On each node a CGI program (called a "TaskApp") is launched with a unique set of parameter values that define a single partition of the overall task. Hence this system implements the single-program, multiple-data (SPMD) parallel processing model. Computations are managed on a kernel server running a set of kernel modules, namely Dispatcher, Status, Statekeeper, Collector, and DataManager. A Web browser interface is used to interact with these modules to administer nodes and computations. MoBiDiCK is back-ended by a database that holds state information about nodes and tasks. The MoBiDiCK kernel runs on the HP N-class server, the cluster head.

A computation is started with the Dispatcher module to partition and dispatch the task. The Collector handles output collection and local node cleanup, while the Status and Statekeeper modules perform task monitoring and fault-tolerance functions. Since there is no need to directly log into a node to run a task, user accounts are not required on the cluster nodes. As with any CGI program, a MoBiDiCK TaskApp is launched by the HTTP server under pre-configured restricted user and group IDs (usually the “nobody” user and group on Linux). We configured the cluster nodes to run the Apache Web Server[9] using a modified “TimeOut” directive to allow TaskApps to execute beyond the 5 minute default value.

4. Cluster applications

The cluster computer is capable of running a variety of software optimized for Beowulf clusters using industry standard systems (PVM, MPI) for controlling parallel applications. This includes software important in our field of research for studying molecular dynamics (CHARMM) and for the production of high-resolution 3-D ray-traced graphics (PVM-POVRAY). We have already described our main application, FOLDTRAJ, which is used for sampling the 3-dimensional conformational space available to a protein structure. Here we present a new application for exhaustive sequence comparisons.

4.1. MOBLAST – parallel M×N BLAST with exhaustive alignments

We report a new cluster application aimed at providing exhaustive protein sequence alignments in an M×N comparison. We approximate using a figure of 500,000 non-redundant sequences, rounded up from the current database size. About 10,000 of these sequences have known 3-D structures. We are interested in obtaining a database of the pre-computed comparison of all sequences against each other, to serve as a resource for a variety of other computations. This is not difficult to compute on a cluster of this size with the currently distributed BLAST application for protein sequences, *blastpgp*, which is supplied in the NCBI toolkit. However we are interested in a more detailed computation for sequences with known 3-D structure, an M×N comparison, and we have written a new application for this purpose, called MOBLAST.

We have undertaken to estimate the time and storage requirements for this new computation as we have been building our cluster. This has guided us to find where there are limiting amounts of time and storage as indicated in Table 2. The most ambitious computation of NxN with an exhaustive BLAST method would require almost a year of time and 30 Tb of space to record the output!

The BLAST programs are a part of the National Center for Biotechnology Information (NCBI) programming toolkit. In MOBLAST we use a core routine which performs a pairwise alignment of two sequences, with a function call known as `BlastTwoSeqs()`. In order to speed up the algorithm, we eliminated some overhead I/O by reading the scoring matrix file into memory at the start of execution instead of it being read at every function call. Additionally, we implemented POSIX threading to run the executable on two CPUs simultaneously which optimized memory use on each node by only loading one copy of the database range being searched.

To estimate how much time one cluster node takes per comparison we averaged executions on different database ranges. Each run compared a subset the database range in NxN fashion (only in lower triangle including the diagonal) using threaded MOBLAST on our dual CPU cluster nodes simultaneously. Timing for MOBLAST was estimated from 26 test runs on a database sample size of 500 sequences (in NxN comparison in lower triangle including diagonal). Sequences were sampled from the database, which at the time of the run contained 408,950 sequences, and the database is present on each node’s hard disk at time of execution. The BLAST parameters were left at defaults, which are the same as the *blastpgp* parameters. The “expect” (E) parameter value was 10.0. The average time to compute one MOBLAST comparison on a single cluster node (threaded on two CPUs) was determined to be 0.027 seconds. Overhead does not add significantly to the computation, as it is “embarassingly parallel” in nature. Only 63% of MOBLAST comparisons return a “seqalign” object, of average length 342 bytes in an optimized format. The rest of the comparisons fall under the cut-off value and are not stored.

We also tested the *blastpgp* program directly from the NCBI toolkit in 30 runs. The parameters were BLAST defaults, which include the E value of 10.0. In each run, one sequence (selected by stepping through the entire database over 15,000 sequences at the time) was compared to the NR database (485,275 sequences). The average query length in this trial was 380 characters (the overall average for the whole database is 313 characters). The average time to complete one query was 48 seconds and average number of hits in one query was 1814. It is important to note that the output can be limited by altering the BLAST parameters to cut off less significant hits.

Table 2. Resource Estimates for M×N BLAST Comparisons

Problem	*No. of Comparisons*	*Time on Cluster*	*Output Storage*
1XN (MOBLAST)	500,000	2 min. (estimated)	108 Mb
1xN (blastpgp - heuristic)	500,000	48 sec. just 1 CPU	171 Kb
M×N (MOBLAST; structures vs. sequences)	500,000 x 10,000 (lower triangle)	14 days (estimated)	1 Tb
NxN (blastpgp - heuristic)	$500{,}000^2$	3 days (estimated)	85 Gb
NxN (MOBLAST)	$\sim 500{,}000^2/2$ (lower triangle)	355 days (estimated)	27 Tb

4.2. A generalization for M×N comparisons with MoBiDiCK

MOBLAST is designed to operate with the Modular Big Distributed Computing Kernel (MoBiDiCK). MOBLAST was integrated with MoBiDiCK with the task applications programming interface (Task API). We recognize that other applications in bioinformatics employ MxN comparisons, and therefore sought a generalized framework to partition tasks in that category under MoBiDiCK.

Given a set of query sequences and a set of target sequences, MOBLAST performs a BLAST comparison of each query sequence with each target sequence. Since a sequence is actually a database record, we can express the query and target sets as database ranges Q* and T* such that $Q^*=[q_L, q_U]$, $T^*=[t_L, t_U]$; q_L and t_L denote lower record boundaries, while q_U and t_U denote upper record boundaries of Q* and T*, respectively.

A database can be viewed as a finite ordered set of discrete points along an axis, each point representing a database record. In Figure 2 we illustrate a database using two axes to represent separately the query and target ranges. Note that both axes correspond to the same database. The shaded region consists of all possible pairs of records between ranges Q* and T*, each pair representing a single exhaustive BLAST comparison. BLAST is a commutative operation, so that given two records *a* and *b*, BLAST(*a*,*b*) is equivalent to BLAST(*b*,*a*). If Q* and T* share an overlap range, V*, then each pair in the upper triangle region of V* has an equivalent pair in the lower triangle; we can thus discard all pairs appearing strictly above the diagonal that divides V*. Figure 3(a-c) shows that if Q* and T* overlap, the overlapping range V* can occur in different ways. In each case the overlap range V* can always be divided into equivalent lower and upper triangles along a diagonal.

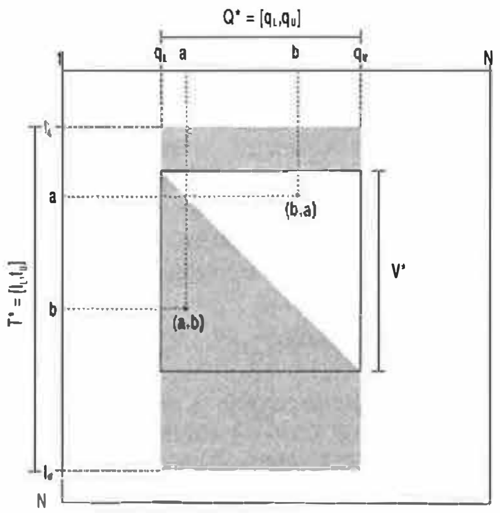


Figure 2. Query and target sets are represented as database ranges. The horizontal axis holds the query range (Q*), while the vertical axis holds the target range (T*). An overlap region (V*) occurs if some records are in both Q* and T*. Record pairs above the diagonal V* are redundant if the operation to be performed is commutative, as is the case for BLAST.

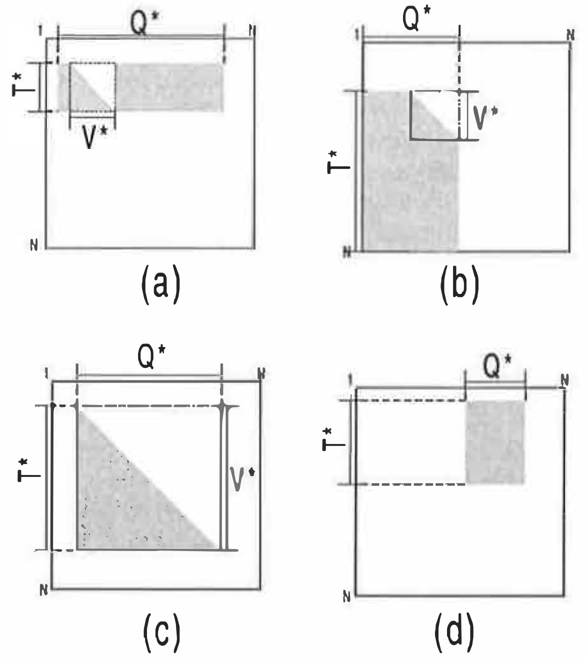


Figure 3. Query (Q*) and target (T*) ranges can overlap partially (a, b), fully (c), or not at all (d). Only record pairs in the lower triangle of the overlap range (V*) are to be compared with BLAST.

We now describe a scheme for partitioning a MOBLAST computation into subtasks that can be distributed using MoBiDiCK. This scheme can serve as an asbstraction that is generalizable to other M×N computations in which a binary operator (such as BLAST) must be applied to record pairs from two ranges of a database. Given a global query range Q* of size M and a global target range T* of size N, we divide both Q* and T* by a partitioning factor *p*, resulting in *local* query ranges Q_1, Q_2, …,Q_p, and *local* target ranges T_1, T_2, …,T_p. as in the example in Figure 4. For simplicity we assume here that both M and N are divisible by *p*. Each cell in the resulting grid represents a distinct MOBLAST subtask that can be performed independently of the others. Partitioning is automatically done by the MoBiDiCK Dispatcher, given values for Q*, T*, and *p*. A total of p^2 subtasks are generated, each subtask being assigned to one node. If the number of subtasks exceeds the number of available nodes, the remaining subtasks that cannot be initially assigned are placed in a queue and are dispatched as nodes become available.

4.3. The M×N algorithm

Each grid cell in Figure 4 represents a subtask of a MOBLAST computation or other pairwise comparison. According to the shape of the shaded region of a cell, we characterize the corresponding subtask as *full* (entire cell is shaded), *partial* (part of the cell is shaded), or *emtpy* (cell is unshaded). All subtasks strictly outside V* are full; subtasks appearing in V* that are strictly below the diagonal are also full. Subtasks completely in V* that are strictly above the diagonal are empty. Finally, all subtasks that intersect V* are partial. Note that if there is no global overlap, all subtasks are full.

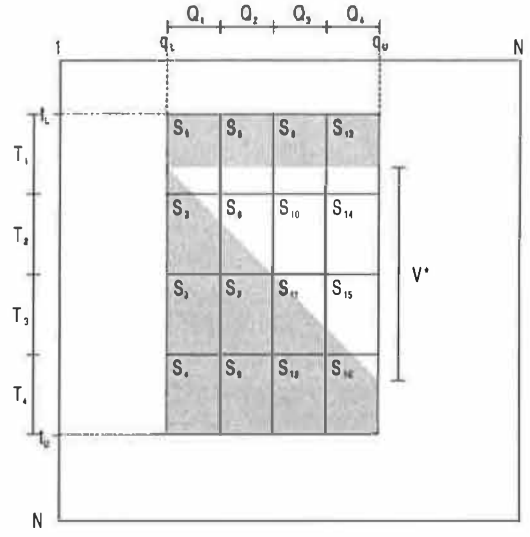


Figure 4. Example of partitioned query and target ranges divided by a partition factor *p* of 4 to produce a grid of 16 subtasks. *Full* subtasks are completely shaded (S_3, S_4, S_7, S_8, S_{12}); *partial* subtasks (S_1, S_2, S_5, S_6, S_9, S_{11}, S_{13}, S_{15}, S_{16}) intersect the diagonal and are only partly shaded; *emtpy* subtasks are unshaded because they appear in the upper triangle of the global overlap region, V*.

We describe the pseudocode for the MOBLAST algorithm given in Table 3. Given global ranges Q* and T* and local ranges Q and T, MOBLAST begins by determining whether the subtask is full, partial, or empty. If neither Q nor T intersect the global overlap range V*, the subtask is full (lines 4, 13). If one of Q and T is in V* (line 4), the subtask is partial only if there is a local overlap between Q and T, that is, if the local overlap range V is non-empty (line 5). If there is no local overlap, the subtask is full if it is in the lower triangle of V*, which occurs only when the lower boundary, T_l, of the target range is greater then the upper boundary, Q_u, of the query range (line 8). If this is not the case, the subtask is in the upper triangle of V* and is thus emtpy, resulting in no comparisons (line 11). BLAST_FULL simply performs a BLAST comparison for every pair between the query and target ranges. BLAST_PARTIAL discards pairs that occur in the upper triangle region of the global overlap range V*.

Table 3. MOBLAST Algorithm

```
MOBLAST(Q*,T*,Q,T)
1       V* ← Q* ∩ T*            ⇨ global overlap range
2       V  ← Q  ∩ T          ⇨ local overlap range
3       if V* ≠ {}                     ⇨ if there is global overlap
4          if (Q ∩ V*) OR (T ∩ V*)  ⇨ we're in V*
5             if V ≠ {}                     ⇨ there is local overlap
6                do BLAST_PARTIAL(Q,T,V*)
7             else                                ⇨ no local overlap
8                if $T_1$ > $Q_u$                        ⇨ in lower triangle
9                   do BLAST_FULL(Q,T)
10               else                             ⇨ in upper triangle
11                  return
12         else                                   ⇨ we're not in V*
13            do BLAST_FULL(Q,T)
14      else                           ⇨ no global overlap
15         do BLAST_FULL(Q,T)


BLAST_FULL(Q,T)     ⇨ compare all pairs
        for i ← $Q_1$ ... $Q_u$
           for j ← $T_1$ ... $T_u$
              BLAST(i,j)


BLAST_PARTIAL(Q,T,V*)      ⇨ only compare pairs outside upper triangle
        for i ← $Q_1$ ... $Q_u$
           for j ← $T_1$ ... $T_u$
              if (i ∈ V*) AND (j ∈ V*)
                 if (j ≥ i)
                    BLAST(i,j)
              else
                 BLAST(i,j)
```

5. Conclusions and future directions

The M×N comparison of proteins with known structures to other sequences creates a database that will allow us to accumulate plausible local structures that may be rapidly assigned to any given sequence, combined with evolutionary information from similar sequences. The accumulation of information about local structure can effectively seed a conformational search procedure and we anticipate that this combined information will expedite a protein folding computation. This would proceed as follows:

- given a query protein sequence *p*
- *S* similar sequences are found using *blastpgp*
- All 3-D fragments *F* matching any subsequence in *S* are retrieved from the M×N database
- A 3-D trajectory distribution *T* is derived from the assembly of *F* mapped through alignments of *S* onto the sequence of *p*
- *T* and *p* form the input to FOLDTRAJ
- Trajectory directed ensemble sampling attempts to predict the 3-D structure

We will describe in more detail this work at our next opportunity. In this work we devised a partitioning scheme for the MOBLAST computation which divides query and target database ranges into smaller subranges. Each pair of query and target subranges can thus be distributed as an independent subtask to a cluster node using the MoBiDiCK distributed computing system. This scheme, however, is not limited to MOBLAST or to MoBiDiCK, and can be generalized to other operations to be performed between pairs of records in a database.

6. References

[1] Altschul SF, Gish W, Miller W, Myers EW, Lipman DJ. Basic local alignment search tool. J. Mol. Biol. (1990) 215:403-410.

[2] Altschul SF, Madden TL, Schaffer AA, Zhang J, Zhang Z, Miller W, Lipman DJ. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. Nucleic. Acids. Res. (1997) 25:3389-3402.

[3] Smith, TF and Waterman, MS. Identification of common molecular subsequences. J. Mol. Biol. (1981) 147:195-197.

[4] Simons KT, Kooperberg C, Huang E, Baker D. Assembly of protein tertiary structures from fragments with similar local sequences using simulated annealing and Bayesian scoring functions. J Mol Biol (1997) 268:209-225.

[5] Dharsee, M. Hogue, C.W.V. MoBiDiCK: A Tool for Distributed Computing on the Internet. In *Proceedings, 9th Heterogeneous Computing Workshop*, May 2000.

[6] Ostell, J., et al. The NCBI Software Development Toolkit. (6.0). ftp://ftp.ncbi.nlm.nih.gov/toolbox/.

[7] Gibrat, J-F., Madej, T., and Bryant, S.H. Surprising similarities in structure comparison. (1996) Curr. Opin, Struct. Biol. 6, 377-385.

[8] Feldman, H.J., Hogue, C.W.V. A Fast Method to Sample Real Protein Coformational Space. Proteins: Structure Function and Genetics, 2000. 39:112-131.

[9] The Apache Software Foundation. Apache HTTP Server Project. http://www.apache.org/httpd.html.

The Development and Integration of a Distributed 3D FFT for a Cluster of Workstations

Dr. Christopher E. Cramer
Duke University
cec@ee.duke.edu, http://www.ee.duke.edu/~cec
Dr. John A. Board
Duke University
jab@ee.duke.edu, http://www.ee.duke.edu/~jab

Abstract

In this paper, the authors discuss the steps taken in the formulation of a parallel 3D FFT with good scalability on a cluster of fast workstations connected via commodity 100 Mb/s ethernet. The motivation for this work is to improve the performance and scalability of the Distributed Particle Mesh Ewald (DPME) N-body solver. Scalability issues in the FFT and DPME as an application are presented separately. Also discussed are scalability issues related to the networking hardware used in the cluster. Results indicate that the existence of a parallel FFT significantly improves performance in DPME from a maximum of 5 processors to at least 24 processors on a cluster of workstations. This has an associated increase in speedup from 4 to 12 times faster than the serial version.

1 Introduction

The Fast Fourier Transform (FFT) has many properties useful in both engineering and scientific computing applications (examples include convolution in real space becoming multiplication in Fourier space, audio compression and every DSP technique known to man). Because of these properties it has become a standard tool in many fields and efficient implementations are a subject of much research interest.

Also of increasing research interest is the subject of Cluster Computing. These clusters of workstations are capable of achieving the performance of traditional supercomputers (on certain problems) at a significantly reduced cost. Such clusters are often loosely coupled groups of machines which communicate using commodity networking hardware. Networking is often 100 Mb/s ethernet, although 1 Gb/s ethernet and proprietary solutions such as Myrinet are becoming more common.

Unfortunately, while there exist many vendor implementations of high speed parallel FFTs for tightly coupled traditional supercomputers, there are few parallel implementations available for clusters of workstations. In our work, we had a need for a ***portable and efficient*** parallel FFT. Not finding one that fit both of our criteria, we decided to implement our own. It was found that the use of this parallel FFT in our code allowed for a great deal of improvement in its performance.

2 Parallel FFTs

The problem with a parallel FFT is that the computational work involved is $O(Nlog_2N)$ while the amount of communication is $O(N)$. This means that for small values of N (we are targeting 64x64x64 3D FFTs), the communication costs rapidly overwhelmed the parallel computation savings. While many tightly coupled parallel processing machines (Cray, SGI Octane, etc) have customized parallel FFT routines, there are few implementations written for a cluster of workstations (i.e. written in PVM or MPI).

Other researchers, when facing this problem, have resorted to using a naive DFT. So long as one is willing to have multiple processors each keep a copy of the space upon which the DFT is to be performed, the DFT is highly scalable in that the results for any given point in the space do not depend on earlier computation that may have been performed on another processor. While this approach can achieve not only speed up, but also performance improvements over a serial FFT, it seems aesthetically unpleasing and if an efficient, portable, parallel FFT is available, it might not be the best use of processor resources.

2.1 FFTW's MPI-based FFT

One of the faster, yet still portable and freely available, implementations of the FFT is FFTW ([FJ99]). Due to it's portability, we have made use of the library in our own applications.

The latest versions of the FFTW library do have a parallel FFT using MPI for interprocess communications. While our application (DPME) is written using PVM, this would not have been a serious impediment as we were already considering an MPI port of DPME. Unfortunately, the FFT did not scale for problem sizes we are interested in (except for very large FFTs, 256x256x256, the speedup was less than 1). This was even true when using the "transposed" FFT which has significantly less communication (see below for a discussion of transposed FFTs).

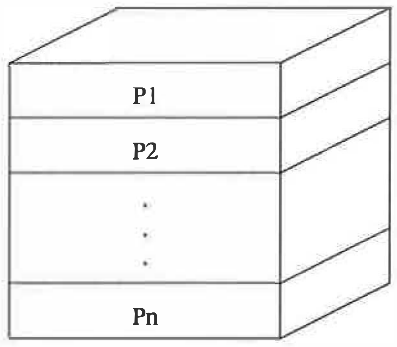


Figure 1: Processor representation of the 3D data space.

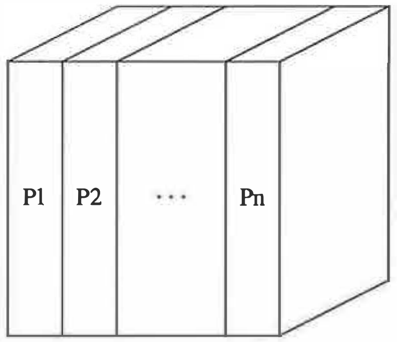


Figure 2: Processor representation of the 3D data space after 2D FFTs and communication step.

2.2 Our Parallel FFT

It was then decided that we needed to create our own parallel FFT which we based on FFTW's sequential FFT libraries. Our FFT was originally written in MPI, but was also ported to PVM for easy incorporation into (the then current version of) DPME. The FFT starts with the assumption that each processor contained one slab of the 3D data space (see Figure 1).

Each slab of the data space contains (for now, exactly) $m = N/P$ 2D slices. Each processor computes a single 2D FFT on each of its m slices, sending the results to all other processors using non-blocking communication. After each processor has received all of the data sent to it, the data is scattered on the processors as seen in Figure 2.

Finally, each processor performs $m * N$ 1D FFTs in the final dimension, yielding the 3D FFT. Note that unless an additional communication step is performed, the resulting data is on different processors than was the original data. This is known as a transposed FFT. In this case, performing either a forwards or backwards FFT results in the Most Significant and the Second Most Significant axes being swapped. So, if the original data layout was stored in ZYX major order, with the data along the Z axis being partitioned to different processors, either a forwards or backwards FFT would result in a changed data layout to YZX major order with the Y axis partitioned to the different processors. The algorithm was designed this way in order to minimize communication, while still being symmetric. In other words, one may call the forward or backward FFT first and have the opposite call put the data back in place.

3 Testbed Configuration

All of the experimental results given in this paper were computed on a cluster of 16 dual processor machines. The processors are Intel Pentium II running at a clock speed of 450 MHz. Each node has 512 MB of RAM as well as an additional 512 MB of virtual memory. The machines are connected using a private 100 Mb/s Ethernet Cisco 2924 switch. All nodes are running the GNU/Linux operating system with kernel release 2.2.12 compiled for SMP.

No kernel patches have been applied to increase network performance as it was found that machines were already capable of 95% of wire speed bandwidth. Kernel patches designed to reduce latency might increase performance slightly at the cost of reducing cluster maintainability.

The parallel programming of this cluster has been performed using both PVM and MPI. PVM ([GBD$^+$94]) results are given using version 3.4.3. MPI ([SOHL$^+$96]) results were found using LAM MPI version 6.3.1.

4 FFT Results

The FFT algorithm described above performs very well provided that one can make use of transposed data in Fourier space. Fortunately, DPME's computation of the electrostatic potential in Fourier space can easily accommodate transposed data. Speed-up results (as compared to a single call to the sequential FFTW 3D FFT routine) for this algorithm are given in Figure 3 and Figure 5 (Figures 4 and 6 give the respective timings). As can be seen from the figures, the method achieves reasonable efficiency for up to 12 processors. The sudden degradation after 12 processors in performance will be discussed in a subsequent section.

The algorithm has also been tested on the IBM SP2 at the North Carolina Super-Computing Center (NCSC). As shown in Figures 7 and 8, the algorithm exhibits moderate speed-up for small FFTs (64x64x64) and good speed-up for medium sized FFTs (128x128x128) - 70% efficiency for up to 32 processors.

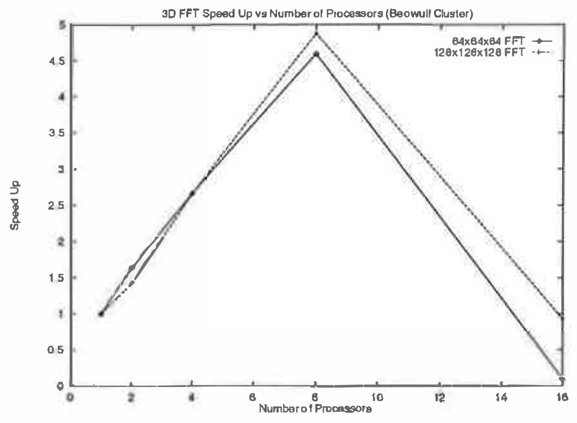


Figure 3: 3D FFT performance on the Duke ECE Beowulf Cluster - speedup

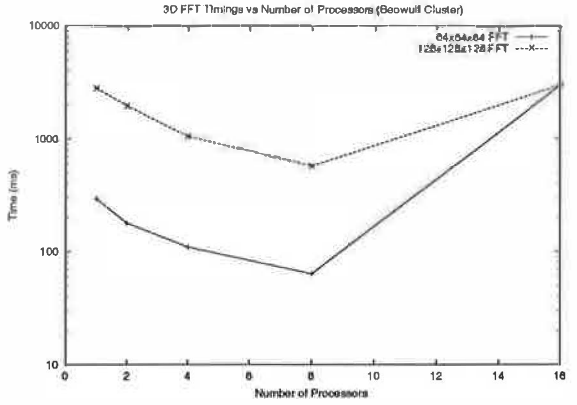


Figure 4: 3D FFT performance on the Duke ECE Beowulf Cluster - timings

4.1 Algorithmic Comparison

The speed up of our algorithm is significantly greater than FFTW's parallel FFT. This is primarily due to the way communication is handled in each algorithm. FFTW has each processor perform all of its 2D FFTs in a single FFTW library call. Then there is a blocking communication step and finally the remaining 1D FFTs are performed. While this decision makes sense in that FFTW achieves its greatest efficiency when performing multiple FFTs, it does not address the real problem in a cluster of workstations: communication. All of the communication in our algorithm is non-blocking. Furthermore, by interleaving communication with computation, we are able to hide a greater degree of the communication overhead involved.

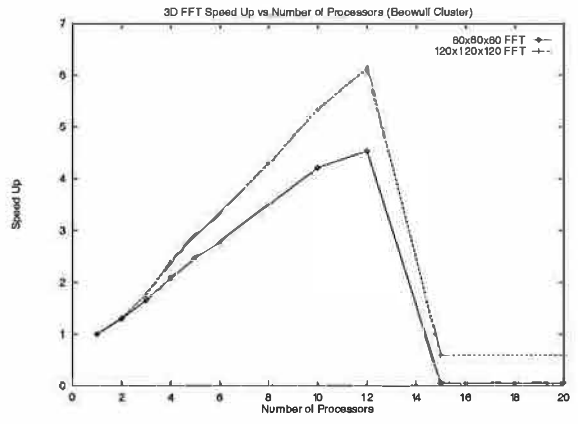


Figure 5: 3D FFT performance on the Duke ECE Beowulf Cluster - speedup

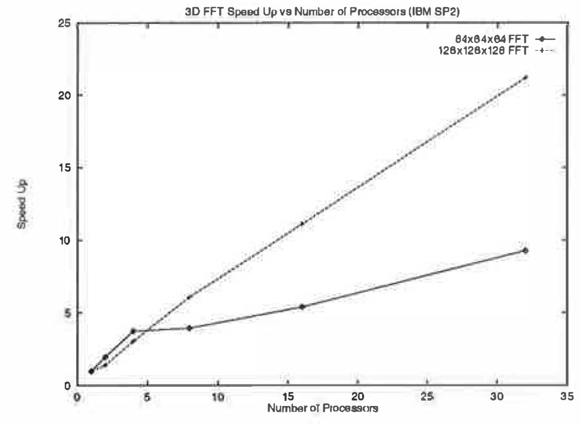


Figure 7: 3D FFT performance on the IBM SP2 - speedup

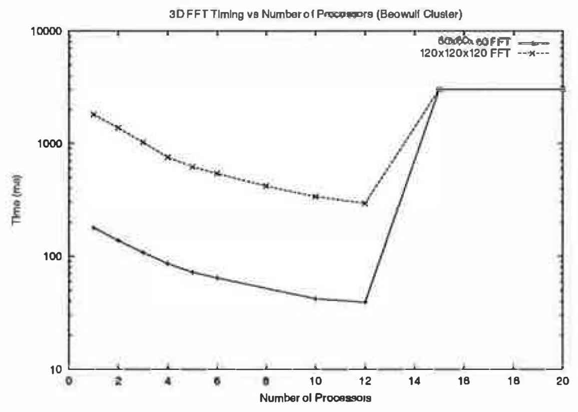


Figure 6: 3D FFT performance on the Duke ECE Beowulf Cluster - timings

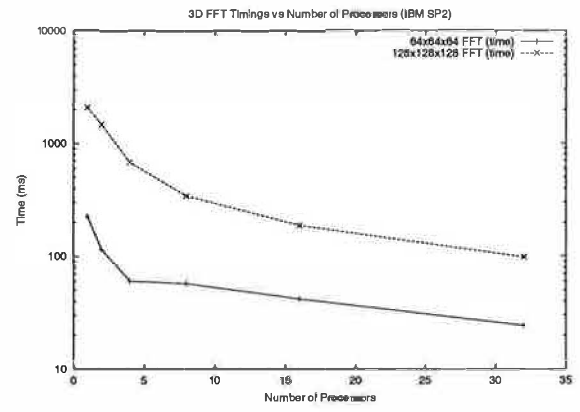


Figure 8: 3D FFT performance on the IBM SP2 - timings

4.2 Hardware Considerations

Early results of our parallel algorithm demonstrated scalability only up to four processors. After that point, speedup dropped significantly. Surprised at the suddenness of the scaling drop-off, we began investigating whether the reduced scalability was due to our algorithm or the specific hardware being used.

At the time, the cluster used an Intel 510T 100 Mb/s switch. It was decided to try replacing this switch with a Cisco 2924. After the replacement, we saw improved scalability of the FFT algorithm up to 12 processors. These results were well correlated with those presented by Mier Communications, Inc ([Mie98]) for the number of simultaneous 100 Mb/s full-duplex streams that could be supported by the various switches without dropping packets. It should be noted that the Mier results were for the Cisco 2916 and the Intel 510T. However, the Cisco 2916 and 2924 have similar backplane architectures.

The Mier Communications tests also included a comparison with the Bay 350. We then tried using a switch in the same family, the Bay 450. Again, the scaling results on our algorithm were well correlated with the number of simultaneous 100 Mb/s full-duplex streams that the switch backplane could handle without dropping packets.

It is uncertain why the switches perform so differently. The backplanes of each switch are all rated at over 2 Gb/s (Intel - 2.1 Gb/s, Bay - 2.5 Gb/s, Cisco - 3.2 Gb/s). Latency also does not appear to be a factor as all of the switches have minimum latency times of $10\mu s$ or less. The most likely reason behind the differences is that the backplane architecture of the various switches results in the Intel switch dropping packets under full load. Since MPI and PVM communicate using TCP/IP protocols, this results in the packet being resent.

Therefore, the strong correlation between the number of simultaneous full-duplex streams (found by Mier) and our own scaling results is likely due to the nature of our algorithm which aims to fully utilize the switch architecture by having each processor send data to every other processor after each 2D slice of the data has been converted to Fourier space. So, each processor is not only sending data to all other processors, but is also receiving data from each of the other processors. The amount of data in each message depends on the size of the FFT and the number of processors being utilized. In general, it is: $B * N^2/P$ where B is the data size (for the complex doubles we are using, this is 16 bytes), N is the size along one dimension of the FFT and P is the number of processors. For a $64x64x64$ FFT on 4 processors, the messages are 16 kB in size. The number of messages is $N * P$. The number and size of these messages insures that the switch is fully utilized on the P ports. Therefore, the number of simultaneous full-duplex streams capable of being supported is very important.

Determining whether the most important factor in the algorithm's performance is the switch bandwidth or the overall latency is a fairly difficult task. The various switches we used on our local cluster all had similar latencies and had the same (theoretical bandwidths), however, they performed in vastly different manners. The IBM SP2 had a completely different switch architecture, with over 1 Gb/s bandwidth and very low latencies. However, some analysis of the amount of data being sent should give us some idea of which factor (bandwidth or latency) is the most important.

Consider a $64x64x64$ parallel 3D FFT running on 4 processors. Each 2D slice of the data is 64 kB in size. Each message (to the 3 other processors) for this slice is 16 kB in size. On a 100 Mb/s switch, the theoretical minimum time that this message could cross the network is approximately 1.3ms. Switch latencies are on the order of $10\mu s$. When the TCP stack latencies are considered, the latencies involved are still roughly an order of magnitude less than the time to transfer the message. Of course if more processors are used, if the FFTs are smaller, or if the switch bandwidth increases, the latencies inherent in the switch and the TCP stack become significantly more important.

This fact has been recognized by IBM in developing the SP2 in that there are two interfaces for accessing the switch. The standard method routes a user's network requests through the kernel to the adapter and then to the switch. This results in a throughput of 440 Mb/s primarily because of the latency of $165\mu s$. The second method bypasses the kernel and goes directly from user space to the adapter, resulting in the throughput increasing by a factor of 2.4 to approximately 1 Gb/s due in part to the latency being decreased by a factor of 6.9 to $24\mu s$. If we repeat the previous examination of bandwidth versus latency for the two SP2 network interfaces, on the first interface we find that at a speed of 440 Mb/s, each message only requires $297\mu s$ to traverse the switch. In this case, the $165\mu s$ latency is within the same order of magnitude as the transport time, causing latency to be a much greater factor than for our 100 Mb/s switches. For the second interface, the transport time is $123\mu s$ while the latency is $24\mu s$. So, on the SP2 we see that latency is more important primarily because of the maximum speed of the switch itself.

5 Application: DPME

The application for which the parallel 3D FFT was developed is Distributed Particle Mesh Ewald (DPME) which is a parallel N-body solver (with Periodic Boundary Conditions) based on the Particle Mesh Ewald (PME) method developed by Tom Darden [DYP93, EPB+95]. Ewald Summation ([Ewa21]) is a technique for finding the electrostatic potential of particles in an infinite lattice. In the method, a single cell of a crystal lattice is assumed to be infinitely replicated in all dimensions. Furthermore, periodic boundary conditions (PBC) are assumed, meaning that as particles leave the cell from one side, they enter from the opposite replication and so emerge on the opposite side of the cell.

5.1 Ewald Summation

The common trait of all Ewald methods is that they split the original problem space of electrostatic point charges into two separate problem spaces. The first, designated the real (or direct) space part contains the original point charges plus proportional charge distributions centered at the same locations in space as each point charge, with the opposite charge. The second problem space is known as the reciprocal or Fourier space. This space contains only the negative of the charge distributions from the direct space. Therefore, adding the two problem spaces yields the original point charge distribution (see Figure 9).

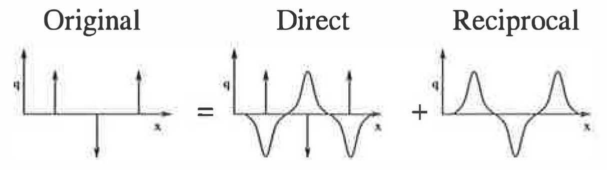


Figure 9: 1D representation of the original, direct and reciprocal problem spaces.

By choosing the appropriate amplitude for the charge distribution, the point charge plus the charge distribution will have an electrostatic potential of 0 at an infinite distance or at any distance where the charge distribution is equal to 0. For infinite distributions, the electrostatic potential can be approximated as 0 at a distance r less than infinity, with a known error bound. Therefore, to solve the direct space portion of the problem, one simply evaluates the all-pairs interaction of each point charge and charge distribution combination with all other combinations centered at a distance less than the cut-off radius.

Due to the periodic boundary conditions, the reciprocal sum is a set of N periodic functions. By solving this problem in the Fourier domain, the infinite periodic functions converge rapidly. The result can then be converted back into real space and summed with the direct space results to obtain the electrostatic potential for the original problem space.

5.2 Particle Mesh Ewald

Particle Mesh Ewald is a derivative of the Ewald method that again splits the problem space into a direct and a reciprocal space. The direct sum is solved by directly computing the interactions between all particles within each particle's cut-off radius. Assuming constant density of the particles being simulated, this is an O(N) problem.

To solve the reciprocal sum, one first discretizes the problem space as a 3D mesh. The charge functions are then interpolated onto the mesh. A 3D FFT is then performed on the mesh, transforming it into Fourier space. The electrostatic potential of the the mesh is computed (still in Fourier space). The mesh is transformed back into real space (by means of an inverse FFT). Finally, the electrostatic potentials of the original point charge locations are interpolated, based on the values in the 3D mesh.

The direct and reciprocal space electrostatic potentials can then be summed to find the electrostatic potential of the total problem space at the locations of each point charge. As was previously mentioned, solving the direct space portion of the problem is O(N). The reciprocal sum's order of complexity is: $O(Mlog(M) + p^3 * M)$ where p is the order of interpolation when converting to or from the mesh and M is the number of mesh points. The 3D FFT is $O(Mlog(M))$ and the interpolation is $O(p^3 * M)$. If M is approximately N, then the complexity of computing the reciprocal sum is $O(Nlog(N))$.

5.3 Distributed Particle Mesh Ewald

Distributed Particle Mesh Ewald (DPME) is a distributed implementation of the PME method, written by Abdulnour Yakoub Toukmaji for his Ph.D. work in Duke University's Scientific Computing research group ([Tou97]).

DPME uses a Master/Slave model for performing parallel computations. The direct sum is performed on the set of Slave processors spawned by PVM. The Master processor performs the reciprocal sum serially. This decision was primarily made due to the lack of a parallel FFT with a speed-up greater than 1 for a cluster of workstations.

DPME has several options which affect performance. The most significant of these is the Ewald Parameter (α) which is inversely proportional to the width of the Gaussian charge distribution. By adjusting the width of the Gaussian, one can shift work between the direct sum and the reciprocal sum. For example, a very narrow Gaussian charge distribution would allow for a small cut-off radius in the direct sum without incurring a large error penalty. However, a narrow Gaussian distribution would require a large number of mesh points in the reciprocal sum. Conversely, a wide charge distribution would result in a nearly uniform reciprocal sum and would therefore require few mesh points, however, the cut-off radius would have to be proportionally larger.

Since the reciprocal sum in DPME is sequential, the obvious strategy to increase its scalability is to set the Ewald parameter relatively small (a wider Gaussian charge distribution). This places most of the work in the direct sum which has a great deal of scalability. Unfortunately, there is a limit to how wide the charge distribution can be. If it is too wide, then the work in the direct sum approaches $O(N^2)$ complexity minimizing the advantages of using a larger number of processors. In general, it was found that optimal performance was achieved using 8 direct sum processors along with the 1 reciprocal sum processor. However, these results were obtained on an older generation of processors. On the cluster we are currently developing under, the performance was significantly worse (see Figure 10).

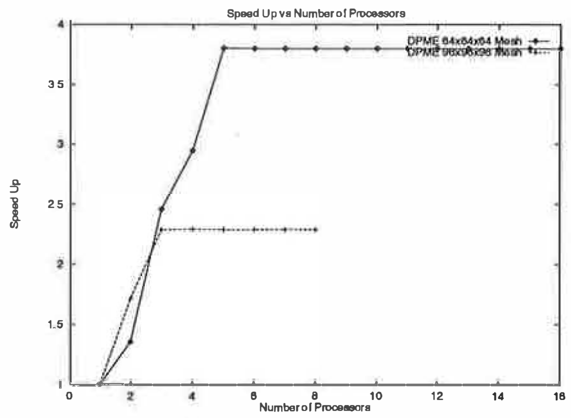


Figure 10: Original DPME speed up curves for 64x64x64 and 96x96x96 meshes

6 Application Speedup

The existence and inclusion of a parallel 3D FFT has allowed us to parallelize the remainder of the reciprocal sum in DPME. This does make it somewhat more difficult to examine speedup in DPME. Before, if you had N processors, then N-1 were dedicated to the direct sum, and one was given to the reciprocal sum. With a parallel reciprocal sum, there can be N-1 ways to divide the work on N processors, since at least one processor must be used in both the direct and reciprocal sum parts. However, since the reciprocal sum is still less efficiently parallelized than the direct sum, it is possible to begin with a single reciprocal sum processor and increase the number until the optimal operating point is found. The following results were computed by taking the desired number of processors and dividing them into direct and reciprocal processors as discussed above until the optimal partition was found. This optimal point is then given as the timing for the number of processors. Performance results for DPME with the parallelized reciprocal sum as compared to the serial reciprocal sum are given in Figure 11 and Figure 12.

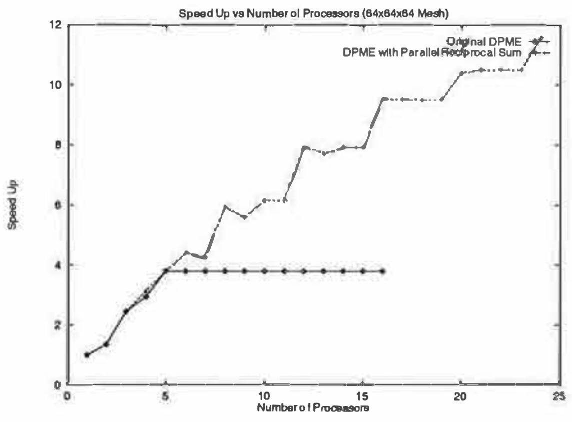


Figure 11: Original DPME speed up versus that of DPME with a parallel reciprocal sum (64x64x64 point mesh)

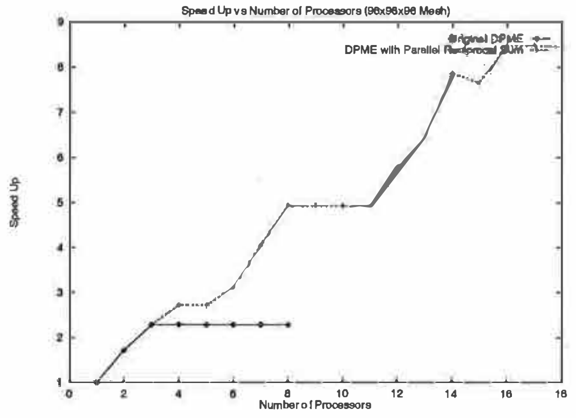


Figure 12: Original DPME speed up versus that of DPME with a parallel reciprocal sum (96x96x96 point mesh)

7 Conclusions

In this paper, we have described the formulation of a parallel 3D FFT capable of achieving speed up on a cluster of workstations connected via 100 Mb/s ethernet. By incorporating this parallel FFT, we have been able to increase the scalability of DPME from a maximum of 5 processors to a maximum of

24 processors, with a corresponding increase in the speedup from 4 to 12.

The FFT itself is general purposed and unlike much other work in the field, it will achieve reasonable speed up on a cluster of workstations. It should give some hope to people contemplating the massively scalable DFT as a replacement for FFT code.

8 Current and Future Work

Currently, we are working on or planning to work on several more improvements in the FFT code for DPME. One area is whether there is an advantage to be had in writing the FFT code to exploit two levels of parallelism on SMP machines: inter-node and intra-node. FFTW does have a thread-based version capable (in our tests) of a speed-up of 1.8 by using two SMP processors on a single node. So, rather than using 8 processes on 4 nodes (8 processors) where each processor had a portion of the data that it operated on, it might prove better to divide the data amongst the 4 nodes. Each node could then put its two processors to work on performing the computation faster. Hopefully, this would shift the operating point of the FFT algorithm to a place where the communication was more efficient (fewer communication calls with more data per call).

Another place where the FFT code might be optimized is in the amount of work done per communication call. As our algorithm stands now, each 2D slice of the data has a 2D FFT performed on it and it is then divided and sent to all other processors. It might prove to be more efficient to perform multiple 2D FFTs before sending the data to the other processors. Of course, if all slices were computed before sending the data, then our algorithm would be identical to the FFTW parallel algorithm. However, there may be an optimal point somewhere between 1 and all slices at a time.

9 Acknowledgments

We would like to thank NIH for funding work on DPME. We would also like to the North Carolina Super-computing Center for grants of time on the Cray T3D and IBM SP2.

References

[DYP93] Tom Darden, Darrin York, and Lee Pedersen. Particle mesh Ewald: An n*log(n) method for Ewald sums in large systems. *Journal of Chemical Physics*, 98(12):10089–10092, 1993.

[EPB+95] Ulrich Essmann, Lalith Perera, Max Berkowitz, Tom Darden, Hsing Lee, and Lee G. Pedersen. A smooth particle mesh Ewald method. *Journal of Chemical Physics*, 103(19):8577–8593, 1995.

[Ewa21] P. Ewald. Die berechnung optischer und elektrostrischer gitterpotentiale. *Anals of Physics*, 64:253, 1921.

[FJ99] Matteo Frigo and Steven G. Johnson. *FFTW User's Manual*, 2.1.2 edition, 1999.

[GBD+94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine, A Users' Guide and Tutorial for Networked Parallel Computing.* Massachusetts Institute of Technology, 1994.

[Mie98] Product lab testing comparison. Technical report, Mier Communications, Inc., 1998.

[SOHL+96] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference.* MIT Press, 1996.

[Swa82] P.N. Swarztrauber. Vectorizing the FFTs. In G. Rodrigue, editor, *Parallel Computations*, pages 51–83. Academic Press, 1982.

[Tou97] Abdulnour Yakoub Toukmaji. *Efficient Methods for Evaluating Periodic Electrostatic interactions on high performance compters.* PhD thesis, Duke University, 1997.

Xfce : A lightweight desktop environment

Olivier Fourdan, fourdan@xfce.org

1) Introduction

There are many differences between UNIX like Operating Systems and desktop workstations' OS that make UNIX and Linux so unique.

One of them is the X layer being independent from the window manager, allowing the end user to choose between more than 60 different available window managers.

Recently, two major projects have emerged on Linux, both trying to reproduce the look and feel of Microsoft Windows and Apple Macintosh environments on UNIX/Linux. KDE and GNOME, as they are known, are doing very well in this, and the developers have done a terrific good job, writing many applications to be embedded in the desktop.

Although these environments work very well on a standalone workstation, they are slow when running across a Network. Moreover, the concept of an X server, with many users connected through the Network does not apply due to network performance issues and memory consumption.

My feeling is that there is a strong need for a lightweight (but still powerful) environment that could run flawlessly when running on small servers and that could serve many X terminals (that can be real X terminals, other UNIX/Linux boxes running X or X emulator running on a Windows workstation).

2) History

Actually the need came first from the simple fact that when I was running one of these environments, there were few system resources left available to run the applications I needed. It was clear to me that the goal was not to run the desktop (as nice as it could be), but to get the job done with the application.

Xfce's primary distinction from other window managers and desktops is its speed. It's designed to run fast. Xfce being more than just a window manager, it's clear that the memory footprint of the whole environment is slightly bigger than running just a window manager. Yet, its memory footprint is much lower than other desktop environments.

The look of Xfce is inspired by CDE, the industry standard desktop on UNIX. But CDE is also very heavy and requires a lot of system resources to run.

The look being close to CDE provides a "professional" appearance for Xfce. Xfce turns Linux into a real, professional workstation according to the users' comments I received.

When I first start coding Xfce, in early 1997, I couldn't imagine how much work it could be and how far it would go. My first goal was to write a toolbar similar to the CDE panel in look (when I first saw the CDE panel, I felt instantly at ease : There were no hierarchical sub–menus to remember; all menus were logically separate and logically arranged by category, etc.). At that time, there was nothing more than the xfce toolbar. It was interfacing with fvwm, using fvwm's module standards.

I first started coding Xfce using T.C. Zhao's Xforms library mainly because Xforms provided a GUI designer and it was very simple to understand and learn. Xfce is entirely written in plain C, with the exception of a few shell scripts and a python script.

After I published the very first version on Metalab (formerly SunSITE), users started asking for some enhancements. I remember the first one was to be able to change the color palette !

Lately, I decided to include my own window manager which was based on fvwm. Actually, I think that the window manager took almost as much time coding as the toolbar itself. I made a lot of changes in fvwm to make xfwm.

The latest and probably biggest change came with the entire rewrite of xfce using GTK+ instead of Xforms last year. There are many reasons for this, one of them is Xforms not being Open Source.

3) Overview of Xfce

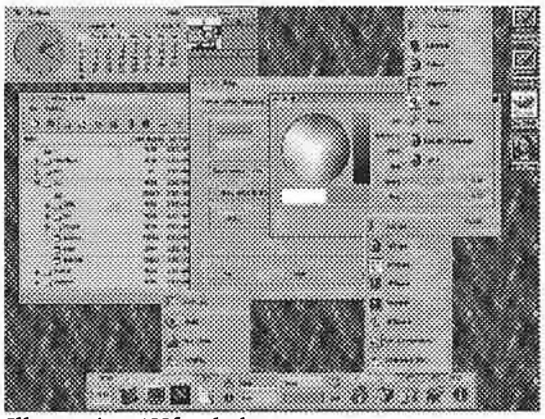
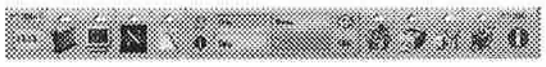


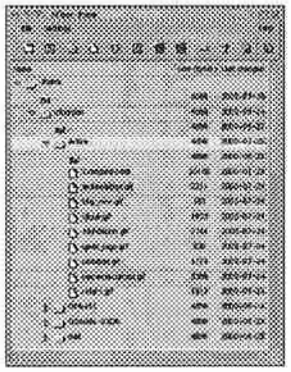
Illustration 1Xfce desktop

Xfce is made of a set of tools that are designed to work together.

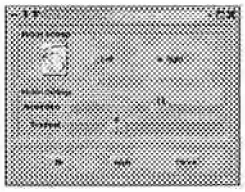
- xfce, the toolbar that resembles the CDE panel :



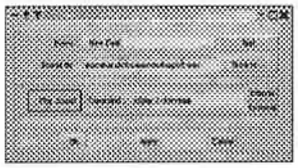
- xfwm, the window manager
- xftree, the file manager :



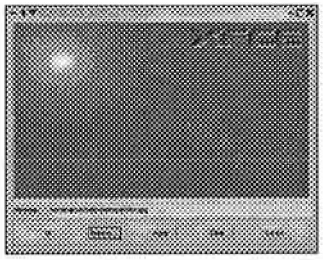
- xfmouse, the mouse settings utility :



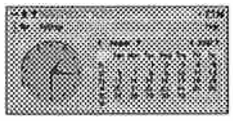
- xfsound, the sound daemon/configuration



- xfbd, the backdrop manager



- xfclock, a clock and calendar



- xfpager, the pager
- xfgnome
- Plus as set of shell scripts for use as Drag and Drop actions for the Xfce panel (xfterm, xftrash, xfprint and xfhelp)

From the Xfce setup dialog, the user can change most of the options for the toolbar and the window manager. This is another strength of Xfce (according to Xfce's users): Xfce does not try to provide the extreme look and feel configurability as some other window managers do, but most of the configuration settings can be changed with a few mouse clicks. Xfce doesn't sacrifice functionality for its small size. The panel is very easy to configure and users can get around without having to constantly use the mouse due to the ample keyboard key bindings available.

Xfce provides the ability to built the color scheme entirely with the mouse, and automatically creates the appropriate .gtkrc configuration file so all GTK applications can share the same look. A small script, written in Python and launched at startup, will manage the color scheme for all X resources so all other standard X applications such as Xt or Motif applications also share the same colors and fonts. This strengthens the concept of a uniform desktop.

4) Usage of Xfce

The configuration and compilation of Xfce are very easy, using standard GNU tools such as GNU autoconf and automake.

Installation is also very easy, as a couple of scripts are provided to easily install/uninstall Xfce on a given user account :

- xfce_setup will replace the current $HOME/.xinitrc file with another one that starts xfce
- xfce_remove will restore user's files
- startxfce can be used as a replacement of startx if one wants to keep their configuration files untouched.

Adding entries to Xfce popup menus is as simple as dragging an executable file from xftree (the file manager) or any other xdnd aware file manager- onto the "add icon" entry.

Changing an existing entry can be done using the right mouse button. This is a general process in Xfce, whenever you want to change an existing icon or a screen label, just click using the left mouse button.

5) Integration with GNOME, KDE and Motif applications.

5.1 GNOME inter-operability

Xfce and GNOME have two things in common : They are both written in C and both based on GTK+, the widget library. It's clear that the GNOME project features tons of different applications for various purpose that Xfce, as a desktop environment, will never cover. Making Xfce and GNOME tightly compatible gives the ability to use GNOME tools transparently with Xfce.

Thanks to the separate GNOME compatibility module (xfgnome) users can turn xfwm into a GNOME-aware window manager and run the GNOME panel along with its applets, including GNOME pager or tasklist.

But Xfce compatibility with GNOME is not limited to the window manager. The xfce panel itself is able to automatically adapt its communication protocol to any other GNOME aware window manager (although when running with xfwm, it doesn't use the GNOME protocol, but rather its specific protocol described later in this document)

This allows the user to run xfce panel with Enlightenment, Icewm, Sawmill or any other GNOME-aware window manager.

Xfwm is able to connect at startup to a standard X11R6-compatible session manager, or fallback to its builtin session management if no session manager is available. As a result, if xfwm/xfce are started from gnome-session, they will automatically interface with it.

5.2 Motif applications

Interactions with Motif applications are limited to window decorations.

Like mwm, the Motif window manager, xfwm is able to read Motif specific hints and adapt window decorations accordingly. GTK+ applications also share the same hints.

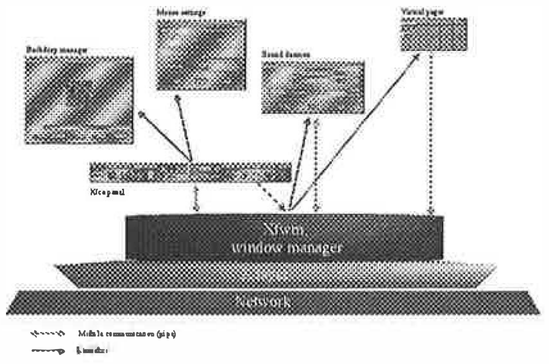
Support for SUN openlook was removed a few months ago, because it was rarely used (except with Openlook applications !). Even SUN has moved to CDE/Motif nowadays.

5.3 KDE

As with Motif, support for KDE applications is limited to window decorations. It means that there is currently no specific KDE module that would allow the use of KDE panel with xfce/xfwm.

But you can obviously use all other KDE applications without problem with Xfce.

6) Architecture



Xfce uses the same techniques as fvwm regarding the communication between modules. Every module communicates with the window manager or other modules using UNIX pipes.
In fact, even the Xfce toolbar can be concidered a module of xfwm.

This provides a high speed communication between modules, but requires the modules to be all spawned by xfwm. Actually, only xfce is really spawned by xfwm, at startup, all other modules are driven by xfce.

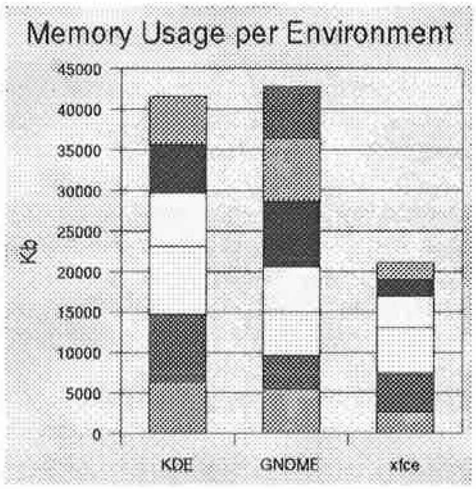
Whenever xfce needs to start a module, from either the startup process or from one of its menus, it calls the window manager asking it to run the module.

This architecture gives more flexibility to the module startup sequence. The user can select from the list of default modules, in xfce setup panel, which module should be launched at startup. This provides the user with the ability to tweak his configuration and reduce memory footprint if some modules are not needed (if you don't use GNOME, you may choose not to start xfgnome, the GNOME compatibility module, or if you don't have a sound card, you'd be better off not to start the sound daemon, etc.)

xfce is designed to save memory. The following charts shows xfce memory footprint compared to GNOME and KDE. To be fair, I've just started the programs that have similar function (window manager, file manager, pager, toolbar, sound, etc.)

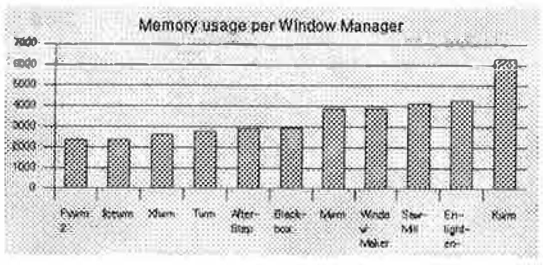
KDE version is 1.1.2 and GNOME is 1.0.55. xfce version is 3.5.0 compiled against gdk-pixbuf (ie without imlib support) :

	KDE		GNOME		xfce
Kwm	6276	gnome-smproxy	5428	xfwm	2616
Kfm	8484	sawmill	4260	xftree	4908
Kpanel	8336	gnome-name-serv	5272	xfce	5516
Kbgmdwm	6596	gnome-session	5600	xfsound	3860
Krootwm	6012	panel	8208	xfgnome	2108
Kwmsound	5932	gmc	7700	Xfpager	2132
		deskguide-applet	6328		
Total	41636	Total	42796	Total	21140



xfwm can also be compared with other window managers available on Linux (the memory usage is given in Kb, all measurement have been made on the same linux system) :

Fvwm2	2320
Icewm	2368
Xfwm	2616
Twm	2772
AfterStep	2904
Blackbox	2920
Mwm	3872
Window Maker	3884
SawMill	4120
Enlightenment	4272
Kwm	6324



7) Future and rewards

The future of Xfce has always been driven from the users' needs and my inspiration! As stated before, when I first started Xfce, I had no idea of what it could be 3 years later. In the same way, I still have no precise idea of what it will be like 3 years from now. Some enhancements that I'd like to implement, though :

Icons on the desktop : This will be a program separate from the file manager.
Modular window manager : By using dynamic linking, I'd like to separate the drawing routines from the core of the window manager. By doing so, xfwm could be able to have different looks without having them coded inside.
KDE compatibility module (similar to xfgnome for GNOME compatibility)
Improve documentation : The current documentation is a user guide in the form of an html page.

I'll probably need contributors for that. I made the biggest part of Xfce, but I would like to take this opportunity to express my thanks to the people who helped and contributed to this project over the years :

Chuck Mead : Without Chuck, xfce would be missing its own web site, its own domain name and the mailing list. Needless to say that Chuck is the man who has helped xfce the most in the past two years.
Joe Klemmer : Joe is the kind of user that any developper would like to have ! He has helped a lot in testing and supporting other users on the list.
And obviously all the people who have contributed patches over the years.

8) Xfce in the press

Linux Planet : *"XFce: The Little Desktop That Could"*

http://www.linuxplanet.com/linuxplanet/reports/1931/1/

LinuxWorld : *"The desktop less travelled"*

http://www.linuxworld.com/linuxworld/lw-1999-12/lw-12-alternative_1_p.html

XFce 3 Initial press release :

http://linuxpr.com/releases/155.html

Linux Gazette :

http://www.linuxgazette.com/issue43/jacobowitz.xfce.html

32Bits Online :

http://www.32bitsonline.com/article.php3?file=issues/199908/xfce/xfce&page=1

Dynamic Probes and Generalised Kernel Hooks Interface for Linux

Richard J Moore - richardj_moore@uk.ibm.com, *IBM Corporation, Linux Technology Centre*

Abstract:

Dynamic Probes (Dprobes)[1] is a generic and pervasive system debugging facility that will operate under the most extreme software conditions such as debugging a deep rooted operating system problems in a live environment. For example, page-manager bugs in the kernel or perhaps user or system problems that will not re-create easily in either a lab or production environment. For such inaccessible problem scenarios Dprobes not only offers a technique for gathering diagnostic information but has a high probability of successful outcome without the need to build custom modules for debugging purposes.

Dprobes allows the insertion of fully automated breakpoints or probepoints, anywhere in the system and user space. Probepoints are global by definition, that is they are defined relative to a module not to a storage address. Each probepoint has an associated set of probe instructions that are interpreted when the probe fires. These instructions allow memory and CPU registers to be examined and altered using conditional logic. When the probe program terminates an external debugging facility may be optionally triggered - should it register for this purpose. For example:

A trace facility may augment its capability with a dynamic trace capability by using the Dprobes facility as a means of inserting tracepoints - dynamically, without any prior code modification.

A crash dump facility may use Dprobes as a means of invoking dumps conditionally when a specific set of circumstances occurs in a particular code path.

A debugger may use Dprobes as high-speed complex conditional breakpoint service.

This paper describes the architecture of Dynamic Probes and briefly discusses a couple of examples of its successful application.

In creating Dynamic Probes, we were challenged with the conflicts between:

Size of the kernel modification
Co-existence with other kernel enhancements, particularly debugging and instrumentation enhancements.
Maintaining concurrency with the latest kernel version.
Ease of development and continued enhancement of Dynamic Probes.

We alleviated these problems by developing a generalised interface for kernel modifications to use allowing them to exist as dynamically loadable kernel modules.

This interface: The **Generalised Kernel Hooks Interface (GKHI)** is described in the second part of this paper.

Historical Perspective

The idea for Dprobes was taken from a technology we previously developed on OS/2[2] originally for implementing tracepoints dynamically without requiring source code modification. This facility operated by making dynamic changes to the code of a loaded module to cause an interrupt at a tracepoint in the same way a debugger inserts a breakpoint. When the tracepoint handler received control it interpreted a small program associated with the tracepoint to collect data from processor registers and memory and built a trace record which was then passed to the system trace facility. Using this methodology there was no overhead when the tracepoints were not active and no requirement to modify a program to allow tracepoints. The trace program contained conditional logic and a limited amount of arithmetical and logical manipulation of data. We extended the programming language to allow Dynamic Trace to invoke user code in the form of device drivers and to exit to other debugging facilities for example:

Kernel Debugger
Application Debugger
System Dump
Application Dump.

Or indeed do nothing! The fact that we had conditional logic gave us a very powerful tool for monitoring a piece of code until conditions presented themselves that required user action. We were also able to accumulate information in the form of local variables, which could be used to retain data from one invocation of a tracepoint to another and be later accessed and display by a command utility.

The original OS/2[2] facility suffered from three design limitations:

1. It was deeply imbedded in large parts of the kernel processing and could not be modularised easily.
2. It was designed primarily as a tracing mechanism.
3. Some simplifications were made upon where and when tracepoints could be placed, for example interrupt-time tracepoints were originally disallowed. Also, because of the mechanism for implementing tracepoints (described later), they could not be placed on certain Intel[3] instructions.

In bringing this technology to Linux we have sought to address these problems in particular. We have attempted to divorce dynamic trace from trace and have generalised it's capability. We have called the Linux realisation of this technology: **Dynamic Probes** or **Dprobes** for short. We have correspondingly changed the term tracepoint to the more generalised **probepoint**.

Over and above the original OS/2[2] idea we have implemented Dprobes with the following characteristics:

It has a greatly extended and generalised probe program command set.

The implementation under Linux has been to make Dprobes an independent facility with a formal interface to allow other debugging facilities to gain control when a probepoint executes.

Dprobes is modular, existing as a set of command utilities and kernel modules. The modification to the kernel has been abstracted to a minimal set of changes by means of a technology we have introduced called **Generalised Kernel Hooks Interface (GKHI).**

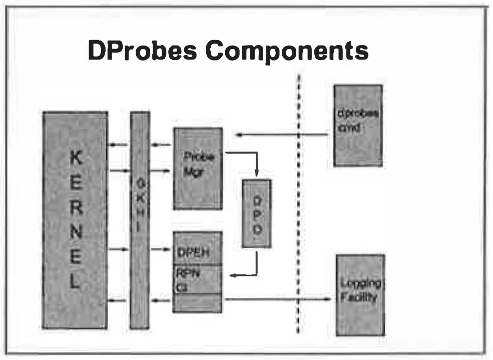
We have also sought to separate platform dependent code from independent code so that the effort in porting to other platforms is minimised.

The probe insertion technique has been improved by delaying the physical insertion of probes in a page of an executable until the time that page is brought into memory on demand.

Probes can be applied to code that is under control of a debugger, without interference to the either the debugger or Dprobes.

Component Overview

Dprobes comprises the following major components whose interrelationships are shown in the figure below:



The Kernel part of Dprobes comprises the Dprobes Manager and the Dprobes Event Handler (DPEH). The Dprobes Manager is responsible for:

Accepting requests to register and deregister probes from the dprobes command line utility. It provides an API for this purpose.

Saving the probe definitions for each probed module in a per-module dynamic probe object[4]. This object comprises the following parts:

The set of probe programs for this module.

The local variable array.

Probe records for each probe defined for that module. Each probe record contains the location of the probe, maintained as file **inode-offset** pair. This provides a universal way of identifying the a probe regardless of whether the module is in memory or not, and if it is, where in memory different instances of the module are located. The probe record also contains a pointer to the probe program associated with it.

As discussed below under **The Breakpoint Mechanism**, probe insertion causes code to be modified in memory. Probes are inserted whenever a page within a probed module is loaded into memory. This is achieved by creating an alias virtual address to the physical address of the probe location . This allows us to insert probe in modules in other process contexts. We also cater for pages marked Copy-on-Write and pages of a shared module that might be loaded at different virtual addresses in different processes. When probes are inserted for the first time we register the **readpage** filter routine for the module. This allows us to be able to re-insert probes when discarded pages are reloaded in memory. This probe insertion technique avoids changing the page state and avoids breaking off multiple copies of swappable pages which would happen if we were merely to store into the virtual address.

Re-inserting probes when a page of code is brought back into memory after having been discarded.

Removal of a probe. As discussed below under **The Breakpoint Mechanism**, we use an instruction-replacement form of breakpoint, which requires us to restore the original instruction in a similar manner to insertion by means of an alias address.

The DPEH is responsible for handling a probe event notification. Event notification occurs when a probed location of a module executes. The DPEH does this by intercepting the system breakpoint and single-step exception handlers. This is described in more detail below. The DPEH identifies a probe event with its dynamic probe object by determining the **inode-offset** that corresponds to the event. It then invokes the **Probe Program Command Interpreter**.

The **Probe Program Command Interpreter** executes commands in the probe program lodged in the dynamic probe object. Should an exception occur then interpretation is quietly terminated with an error indication in the temporary log buffer, which is the temporary piece of storage allocated per processor. This buffer is made available to any external facilities that might register to be notified when the Interpreter exits via the **exit** RPN command. The usable size of the temporary log buffer is determined per probe object, with a fixed maximum size of 1K, which will become configurable.

The user communicates with the kernel components by means of the **dprobes** command line utility, which supports the following major functions:

Insert:

This reads a **dynamic probe definition file (DPDF),** which contains among other things, the module name to which the probe definitions apply, the local variable array size, and for each probe within the module its probe definition. The probe definition comprises the probe location, which may be expressed as a symbolic expression, a user identifier for the probe - a major-minor code pair and the text of the associated RPN probe program. The insert function parses the DPDF and passes a condensed form of this to the probe manager to be saved as a dynamic probe object.
Existing probe definitions for the same target module may be optionally replaced or merged with the new definitions.

Remove:

This will remove all probes from a module. Optionally a subset of the probes may be removed. Probe removal is the reverse of insertion.

Getvars:

This will extract local and global variables for one or more probe definitions. Optionally it may be used to reset the values to zero.

Query

This will display the state of registered probes.

BuildPPDF

This builds a **pre-built probe definition file (PPDF)**, which is essentially a file version of the package built by the **Insert** function. The value in providing this function is that probe definitions can be pre-built from a **makefile** and later inserted using the **dprobes** command's **insert** function. This would allow a module to be installed along with its PPDF so that is debug-ready, without being a special built with debugging code present. Pre-building is made possible because the probe location may be expressed symbolically using symbols from the module's symbol table.

The **dprobes** command supports preprocessor directives supported by the **GNU gcc** compiler. If present, dprobes will invoke **gcc** to resolve preprocessor directives and direct the output to a temporary file against which the probe definitions are parsed. This facilitates:

Substitution into the DPDF from the command line
Macro definitions
Conditional preprocessing

The Breakpoint Mechanism

At the heart of dynamic probes lies the **probepoint** which is a breakpoint - the same as that implemented by a debugger - with a few implementation differences:

Because Dprobes is in one sense an automated kernel debugger we do not wish a breakpoint to interrupt execution temporarily. Instead it gives control to the **Dprobes Event Handler (DPEH)**, which under normal circumstances will return control to the program without user intervention. Because the breakpoint is automated and does not really break execution, we refer to it as a **probepoint**.

As is usual for breakpoints we intercept code before the probed instruction executes. This might seem like an otiose statement given that the beginning of one instruction is the end of another - but not so when the mechanism for implementing the breakpoint is examined. There are in general two mechanisms by which to implement a breakpoint:

Instruction replacement
Watchpoint.

Instruction replacement as the name implies requires that the probed instruction is replaced by another that gives control to the DPEH. Before the DPEH returns control to the probed program we execute the original instruction. Use of instruction replacement provides a pre-execution breakpoint on all hardware platforms.

The watchpoint is a hardware assisted mechanism for interrupting execution in order to give control to some monitoring application, such as the DPEH. Unfortunately watchpoint implementation is not consistent across all processors. On Intel[3] 32-bit architecture there are four debugging registers provided for watchpoint implementation, a severe limitation in itself. Intel[3] watchpoints interrupt on instruction fetch. In contrast S/390[5] watchpoints operate over a continuous range and interrupt on completion or partial completion of the execution of an instruction.

Since there's no easy abstraction of watchpoints across processor platforms we use the *Instruction Replacement* technique.

The DPEH is discussed more fully later however it does play an essential role in the breakpoint mechanism. Its main components are:

Execute *Probe Program* commands
Single-step original instruction
Commit probe buffer
Return to probed program.

On entry to the DPEH, application, system and processor state can be examined as one would do from a kernel debugger but by means of commands in the associated RPN Probe Program. This requires that the DPEH operates at a supervisor level of privilege. Therefore to enter the DPEH from any privilege level we require the breakpoint to be implemented using an instruction that will cause an interrupt. Under Intel[3] the **INT3** instruction suffices and is designed for this purpose. Under S/390[5] the **SVC255** instruction provides an equivalent function. Each processor platform that distinguishes privilege levels will offer an instruction equivalent to these that will allow controlled access into a supervisor privilege level of execution.

The second stage of the DPEH is to single-step the original instruction followed by a commit phase. (Committal applies to the temporary log buffer, when passing this to an external facility, for example a tracing facility or the default **Syslog** facility).

Two questions arise:

1. Why single-step?
2. Why commit after single-stepping?

Because the DPEH is privileged we cannot easily execute a copy of the original instruction in-line in the context of the DPEH since that would:

a. Grant to the original instruction and hence the probed program a privilege level that may not authorised
b. Alter the execution outcome of some instructions, e.g. POPF on Intel[3].

The DPEH allows the Probe Program to save data in a temporary buffer before committal on completion of the single-step. In this way the DPEH provides a tracing or logging mechanism. Some instructions might be interrupted several times and restarted before they complete (I am including here the possibility of recoverable exceptions, for example page-faults). We would not wish to record multiple events for each re-execution. This requires the DPEH to commit log data after the original instruction has completed execution.

Single-step allows the DPEH to execute the original instruction in its intended context and gain control on completion.

In the current implementation under Intel[3] 32-bit architecture (IA32), the single-step phase of the DPEH comprises restoring the original instruction, returning from the DPEH using the **IRET** instruction with the **TRACE** flag set in the **EFLAGS** register. If the single-stepped instruction completes without interruption or exception control is returned to the DPEH via the single-step system exception handler and logged information is committed. The **INT3** is restored and the DPEH returns to the probed program.

If the single-stepped original instruction terminates with an exception (other than single-step), the log buffer is discarded and the **INT3** is restored. This requires that the DPEH be given control from all system exception handlers. (In the current implementation under IA32 we have disallowed probes on **INTn** instructions since these always end in an exception and would require an unnecessary intrusion into the system exception handlers for exceptions 0x20 - 0xff).

Implementation details of the probepoint, particularly the single-step are clearly processor dependent. However, mechanisms for single-stepping instructions under program control exist on all modern processor architectures. The single-step mechanism is therefore customised per architecture. To ease porting to other platforms, the single-step is made modular and therefore easily replaceable. The single-step implementation under Intel[3] more or less forces us to do this with interrupts disabled, since there is no easy way to save state across the single-step should we re-enter the breakpoint exception handler, consequently we single-step with interrupts disabled and make appropriate adjustments for instructions such as **PUSHF**, **CLI** and **STI**. In addition we take steps to avoid recursion due to probepoints being placed in the path of the DPEH. We do this in two ways:

> Preventatively: by disallowing registration of probepoints within the DPEH module.
>
> Reactively: by using a recursion counter to detect unexpected recursion. This caters for probepoints in subroutines called by the DPEH. If we detect recursion we *silently* remove the recursing probepoint and return to the probed code.

The *Instruction Replacement* form of breakpoint has two undesirable side-affects, which may or may not be troublesome, depending on architecture:

In order to single-step the original instruction in context, we temporarily replace the breakpoint instruction with the original instruction. This opens a window of opportunity for a probepoint to be missed if the same piece of probed code is executed on another processor in close succession. We can avoid this by forcing processor serialisation during the single-step. However, that can badly affect performance, and so, is left as an option for the user to deploy if needed. In practice this has not been a problem because Dprobes has been used to find bugs in code that is already serialised or races against other code that jointly accesses common data.
The other side-affect is very much architecture dependent. When dynamically changing instructions, some architectures will require additional actions to be

carried out in order to guarantee consistent results. For example, not all architectures fetch whole instructions as an atomic entity nor do they do this in address order. Furthermore the problem is compounded when code is stored in read-only memory and an update has to be done using an aliased read/write virtual address. Usually there are ways around these problems; one needs to read very carefully the processor documentation regarding instruction caching, pipe-lining and speculative execution.

The instruction-replacement form of breakpoint and single-step requires that the breakpoint be re-instated on completion of the single-step.

The Dynamic Probe Event Handler

The Dynamic Probe Event Handler (DPEH) is invoked as a result of a probepoint executing. As mentioned in the description of the breakpoint mechanism, the DPEH comprises 4 phases of operation:

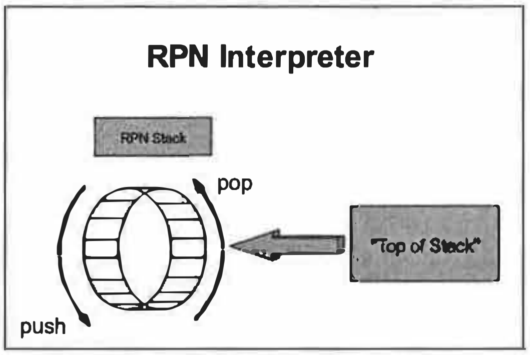
Execute *Probe Program* commands
Single-step original instruction
Commit probe buffer
Return to probed program.

The last three phases were discussed under the previous section - The Breakpoint Mechanism. We now look in detail at the first phase - **The Probe Command Interpreter**.

When a probe is registered with the Dprobes Manager an associated **probe program** is lodged with he Dprobes Manager. This program defines the actions that will be carried out when the probe is executed. The probe program language is of the Reverse Polish Notation (RPN) family of languages. Therefore, an implicit stack is associated with the probe program for temporarily storing operands and results of RPN commands.

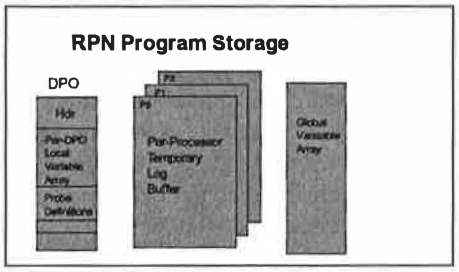
The RPN family of languages are implemented using a stack on which command operands are pushed. When a command is executed the operands are popped from the RPN stack and the result pushed on to the stack. Many implementation use a circular array of fixed width for the RPN stack and maintain a special pointer that locates the Top of Stack (TOS), which is precisely the implementation within Dprobes. Two special (families of) commands are provided to for the program to access the stack:

PUSH
This rotates the stack forward and copies data to the TOS.
POP
This rotates the stack backward, but does not update tack contents.



Dprobes implements a 32-element circular RPN stack of width equal to the bus width of the processor. In the case of Intel[3] 32-bit architecture each RPN stack element is 32-bits.

There are three other storage areas provided for use by the RPN program:



The Temporary Log Buffer is used to accumulate data during the execution of the RPN program. Depending on how the program terminates, will be passed on to an external debugging facility, for example, **Syslog** or **Trace**. The temporary log buffer contents persists only during the execution of the probe program. Contents are discarded if the program ends abnormally or is deliberately aborted. One temporary log buffer is defmed per processor, the usable size of which is defmed per module, when probes are registered.

The Local Variable Storage Array is provided per probed module and shared among all probe programs that are operative for a probed module. Each array element, or local variable, is retained across invocations of the RPN program and may be used to share data between probe programs for a given module. Local Variables can be extracted, displayed and reset by the dprobes command utility. The size of the local variable array is defined when probes for a module are registered.

Global Variable Storage is similar to Local variable Storage, but is common to all probed modules.

The **DPEH RPN Command Interpreter** implements the following classes of commands:

Execution and Sequencing Group.
This group includes conditional jumps, loop, subroutine call, exit control and probe control.

Logging Group.
This group includes commands that update the temporary log buffer.

Local and Global Variable Group.
This includes commands that perform allocation of global variables and command that read and write to global and local variables.

Arithmetic/Logic Group.
This includes commands that operate with the RPN stack. They include addition, multiplication, subtraction, bit masking, bit shifting and rotation.

Address Verification.
This in a small group of two commands that will test the validity of a memory addresses. These are provided because the interpreter runs with interrupts disabled and cannot page-in swapped or discarded memory - see **Data Group** below.

Stack Manipulation.
A single POP command used to rotate the RPN stack without data manipulation. Commands that manipulate data on the RPN stack are included in the groups relating to the data origins and destinations.

Register Group.
This include commands to push processor registers onto the RPN stack and commands to pop registers from the RPN stack. Both the current processor registers and the current user context registers may be accessed.

Data Group.
This includes commands to push and pop data in memory to and from the RPN stack. A subset of this group is the system variable subgroup which will access key system data values, for example current process ID and address of current **task_struct**. Because the DPEH runs with interrupts disabled, exceptions caused by data group commands are soft-failed by the interpreter by halting interpretation and storing a failure code in the temporary log buffer.

IO Group.
This include commands to push and pop data from the IO address space (not implemented on platforms that do not support an IO address space).

Example Probe Programs

Example 1 - fork and kill:

```
name = bzImage
modtype = kernel
major = 1
jmpmax = 32
logmax = 100\\
dfltexit=1
vars = 2

offset = kill_proc
opcode = 0x55
minor = 1
pass_count = 0
max_hits = 1000
inc lv,0
push d,16
push r, esp
log mrf
exit

offset = do_fork
opcode = 0x55
minor = 2
pass_count = 0
max_hits = 1000
inc lv,1
push d,16
push r, esp
log mrf
exit
```

The example above shows an RPN probe that will create a Syslog entry ever time a process forks and is killed. It will accumulate the number of forks and kills in local variables 1 and 0. These variable may be displayed at any time using:

dprobes -g -a

from the command line. Whenever **kill_proc** or **do_fork** is entered, the probe programs above will write to the temporary log 16 bytes of current stack data, which on exit will be written to **Syslog** using **printk**.

Example 2 - malloc:

```
name = "/lib/libc.so.6"
modtype = user
major = 1
jmpmax = 32
logmax = 100
dfltexit=0

offset = __malloc
opcode = 0x55
minor = 1
pass_count = 0
push r,esp              // push the stack
pointer
push d,4
add                     // TOS -> first parm
to malloc (size)
push d,0x20000000       // size 0x20000000
sub                     // compare
jz take_dump            // if equal, take core
dump
exit                    // else exit without
further ado
take_dump: exit 3
```

In this example we place a probe on entry to the **malloc** routine of **libc**. We look for the instance where malloc is called with a size value of **0x20000000** and when found we force a core dump.

Real-life Examples of Dprobes Use.

Dprobes is a new technology for Linux so the number of example problem determination uses on Linux is limited, however it is worth mentioning that Dprobes was used to debug itself during development.

The following examples have been taken from the OS/2[2] platform from which Dprobes was developed. I'll summarise two contrasting examples: the first an operating system bug, the second an application space bug.

Example: A deep-rooted operating system bug.
We had a situation where we found the file-system was page-faulting unexpectedly. What was very odd about this situation was that the page-fault was occurring on a page that should have been locked in memory. We could never reproduce this problem in the lab, it only happened on a customer's server, once a day at peak load.

The first hypothesis was that the file system had a bug and had accidentally unlocked a locked page. So we created a PPDF that contained probe definitions of the file system's locking and unlocking subroutines. We sent this to the customer and asked him to insert the probes. When the system next crashed, the customer sent the crash dump and log created by dprobes.

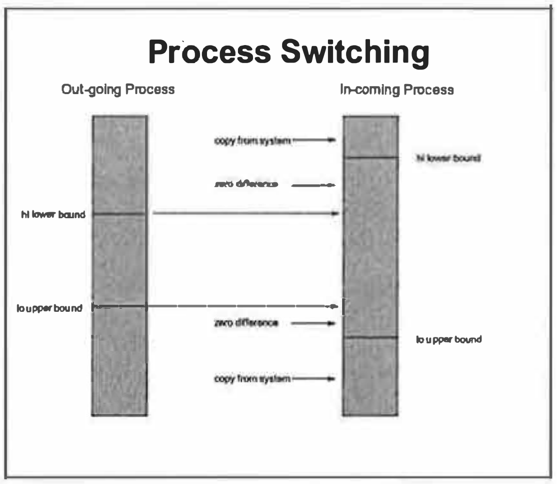
The dump and dprobes logs showed only a lock request for the faulting page, however the page status could be seen from the dump to be unlocked.

The second hypothesis was to assume that another kernel module was unlocking the page. So we sent the customer a PPDF that logged all calls to the kernel's lock and unlock routines. Once again the log showed that no-one had unlocked the faulting page. However we did notice that every time the locked page faulted a great deal of process switching had taken place. We began to suspect that there might be a page-manager problem in the kernel. To see whether something odd was happening we placed probes inside the process context switching routine.

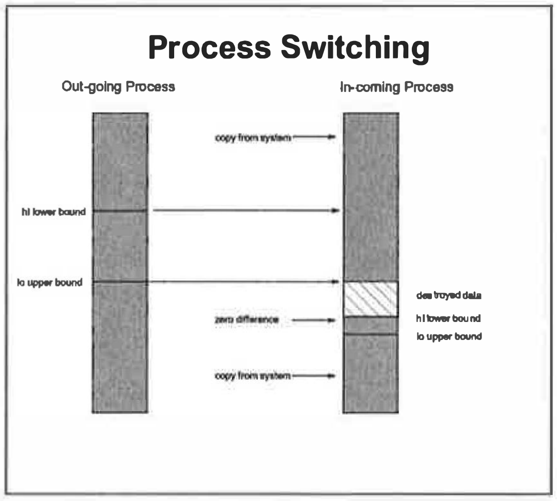
Code inside the context switcher is very difficult to debug, We are in no defined process state. Page Tables and system state variables in an inconsistent state with respect to each other. But because the DPEH was designed to operate in the most hostile conditions we could use Dprobes to find out what was happening. Inside the Context Switcher we placed two probes: one before page data was saved. This probe logged the out-going processes page tables. The other probe was placed near the exit of the context switcher and logged the in-coming process' page tables.

On re-creating the problem we could see that the out-going process' page tables showed the faulting page present and when the process was next run the faulting page table entry was zero. But the copy of the page data, maintained by the system while the process was not being run , still had the page table entry intact (we could see this from the dump). There had to be a bug in the context switcher - despite having been written some 15 years ago and not had a defect in all that time.

It was time to examine the code: someone had introduced a performance enhancement to avoid double-updating of page tables. This enhancement had a bug which caused valid page tables entries to be zeroed when the outgoing process' upper bound for low memory over-lapped the in-coming process' lower bound for high. See the picture below for clarification.



This situation worked until the following occurred:



Example: who sent the parcel-bomb?
This is an application-space example. The messaging facility within OS/2[2] has an asynchronous function: **WinPostMsg**. When a message is posted the application's message thread is posted - made ready to run. When it runs it will find one or messages on its message queue along with optional parameters. A posted message is asynchronous. By the time the receiver wakes, the poster could have even terminated. And there's no way of knowing who it was.

We had a situation where a message was being posted to an application but every now and then a message was posted with an erroneous parameter that caused the receiver to trap. The question is who sent it?

We found this out simply by putting a probe on the entry to the WinPostMsg. The probe contained conditional logic to examine the message ID and the parameter. When the ID and Parameter matched the values that caused the receiver to trap we invoked a crash dump. The dump was taken in the context of the poster and allowed us to see who it was and where in their code it was happening.

Conclusions:
Both these examples could have been solved with other debugging tools, but not easily so. Both would have needed an ability to place global breakpoints at certain code locations and at the same time exercise conditional logic. Potentially a kernel debugger can do this. However, the breakpoints deployed require performance to be maintained, furthermore kernel debuggers tend to assume a breakpoint really means break - so are designed to perform serialisation function to allow direct user communication. The DPEH, on the other hand, is designed with minimal serialisation dependencies and no user interaction. It can therefore maintain system performance with complex conditional breakpoints applied.

Other uses of Dprobes has been to provide a high-speed conditional breakpoint facility which gives control to a kernel debugger when the correct situation presents itself.

Generalised Kernel Hooks Interface

Like other modifications to the base operating system, there are problems with having to manage large modifications, for example:

> If more than one independent enhancement needs to co-exist with another, then patches for each may conflict and have to be resolved - possibly by the user.
>
> If the enhancement needs to be updated then a new patch must be supplied and integrated with existing patches. The kernel will require recompilation and possibly also some kernel modules.
>
> If the kernel needs to be modified for maintenance reasons then the patches need to be re-worked and re-applied.
>
> If the user wants to use patches, albeit infrequently, they must either run with that additional function even when not needed , or switch kernels when the function is needed. Either way there's an inconvenience and an overhead implied.

Dprobes, like other diagnostic and instrumentation facilities tends to insert additional function rather than replace it. For this category of kernel enhancement we have the opportunity to separate the kernel enhancement from the kernel by confining it to a loadable kernel module - provided - interfaces are added to the kernel to allow kernel modules to be called at the appropriate time.

A particular problem that needs to be overcome is how to define such an interface. We cannot simply code within the kernel, calls to external modules because we would not be able to resolve those calls at kernel load time. To overcome this , we define **hooks** within the kernel.

A **hook** is a small piece of code inserted into the kernel source, actually a one-line change because we define the hook using a C macro, an example of which is shown below:

```
#define GKHOOK_2VAR_RO(h, p0, p1) asm volatile
(".global GKHook"h";
                        .global GKHook"h"_ret;
                        .global GKHook"h"_bp;
        /* replace with nop;nop;nop; to activate */
                        GKHook"h": jmp GKHook"h"_bp;
                                    leal %1,%%eax
                                    push %%eax;
                                    leal %0,%%eax;
                                    push %%eax;
                                    push $2;
 /* replace with jmp GKHook"h"_dsp when active*/
                                    Nop;nop;nop;nop;nop
                        GKHook"h"_ret: add $12,%%esp;
                        GKHook"h"_bp:;"
                        ::"m"(p0),"m"(p1):"%eax")
```

The hook begins with a jump to the end of the hook, in its dormant state. The body of the hook contains space for instructions to be added when the hook becomes active. When a hook activates, the body of the hook is populated with instructions to call the **Generalised Kernel Hook Interface (GKHI)** module at the hook's dispatcher entry point and the initial jump instruction is nullified by replacing it with NOP instructions. The GKHI is responsible for activating hooks and makes these modifications when another kernel module calls the GKHI to request that a hook be enabled for its use.

This implementation requires that a minimal change be made to the kernel: a hook be identified at each location in the kernel source where external modules may wish to gain control. But, it allows multiple modules to access the same hook without further modification to the kernel as we shall see when the GKHI is described in more detail.

The GKHI provides four interfaces for kernel modules to call in order to access or relinquish access to one or more kernel hooks.

GHK_register:
This is used to identify a location (a **hook exit**) in a kernel module that wishes to gain control at a given hook. The caller passes a hook registration record to the GKHI which contains the hook exit address and a flag indicating the desired dispatching priority which may be:

- First in list of exists for this hook
- Last in list of exits for this hook
- Only exit for this hook
- Unspecified.

A hook exit will not be dispatched until it is armed.

GHK_arm:
This allows one or more hook exits to be made dispatchable. The kernel module calling this interface will pass a list of chained hook registration records that are to be armed. If for any of the hooks referenced this is the first instance of the hook being used the GKHI will activate the hook. So, it is not until a module needs to use a hook will the additional code path be imposed on the kernel at the hook location. Arming is carried out in an atomic fashion, under SMP this requires other processors to be temporarily suspended from performing useful work. The impact, however, is minimal, since arming amounts to activating the hook if not already active and updating status maintained with in the hook registration record for each hook exit being armed. This is a CPU-bound operation on resident memory.

GKH_disarm:
This has the opposite affect of GKH_arm: a list of hook registration records is passed to the GKHI, which will mark each one as no longer dispatchable. If any one of these is the last to disarm for a given hook then the GKHI will make the hook inactive by restoring the original bypass jump instruction. Disarming is carried out atomically.

GHK_deregister:
This has the opposite affect of GKH_register where each hook exit in the list of exits requiring de-registration will be removed from knowledge of the GKHI.

When a hook is active and exits are armed, the dispatcher routine will be called when the hook executes. This routine is responsible for calling each of the armed hook exits for that hook, in priority order. If any exit returns a non-zero result no further exits are dispatched on that invocation of the dispatcher. As part of the registration of a hook exit, entry conditions may be specified:

All CPUs are temporarily stopped
Interrupts are disabled

In addition, the exit may adjust the status flag in its registration record so to disarm itself. This together with the ability to specify entry conditions gives the exit the an opportunity to be dispatched and disarmed as an atomic entity.

Part of the definition of a hook is whether it will pass any parameters to the hook exit and in addition whether those parameters will be modifiable. In the example given above the hook macro defines two read-only parameters. But in addition to passing these parameters we also pass a count of the number of parameters. In this way a hook may be modified to have additional parameters added. And the exit will know whether it is compatible with the current hook definition. (By convention we add new parameters to the end of this list).

GKHI may be loaded at any time using **insmod**, however no hooks can be activated until the GKHI is loaded. It is possible for kernel modules to define hooks within themselves, but using the regime described so far would require that the kernel module containing the hook be loaded before the GKHI. To achieve even greater flexibility, the GKHI defines two further interfaces:

GKH_identify:
This interface is called to notify the GKHI of a new hook that has become available since it initialised. Obviously exits cannot register for the new hook until it has identified itself.

GKH_delete:
This interface is called to notify the GKHI the a hook is not longer eligible for registration.

Where to obtain Dprobes:

Dprobes, including the GKHI, is available from IBM's Linux Technology Centre's web page at:

http://oss.software.ibm.com/developerworks/opensource/linux/projects/dprobes

The development team comprises:
Richard J Moore (Dprobes Project Lead) - richardj_moore@uk.ibm.com
Bharata B Rao - rbharata@in.ibm.com
Subodh Soni - ssubodh@in.ibm.com
Maneesh Soni - smaneesh@in.ibm.com
Vamsi Krishna Sangavarapu - r1vamsi@in.ibm.com
Suparna Bhattacharya - bsuparna@in.ibm.com

Notes and references:

[1]Dprobes website: http://oss.software.ibm.com/developerworks/opensource/linux/projects/dprobes
[2]OS/2 is a trademark of International Business Machines Corporation.
[3]Intel is a trademark of the Intel Corporation
[4]The dynamic probe object is a complex set of structures, the principle being dp_module_struct. Each of these is defined in **dprobes.h**
[5]S/390 is a trademark of International Business Machines Corporation.

Knowing When To Say No

Allan Cantos
CTO, Acrylis, Inc.

You see a new piece of software posted on freshmeat, and you just know that it will solve that nagging problem you've been having with (insert name here). But what many of us don't realize, until it's far too late, is all the possible ramifications of installing (or deleting) even one simple piece of software. Allan Cantos, CTO at Acrylis (www.acrylis.com) will explore many real-world scenarios and pose questions that every Linux Administrator should ask prior to hitting the download button. Things like how much work is involved if I install that new software? What, if anything, will break in the process? Do I need to update any existing libraries on my system? What other key applications rely on those libraries? In the end, is it worth my time to do an upgrade?

In this seminar you will learn how to:

Understand Recursive Dependencies

While RPM gives some data on dependencies, what dependencies do the dependencies have? A detailed look in recursive dependencies of packages will be explored.

Manage Software Across All Systems

If a security alert on a particular piece of software arises, how can you quickly find it on all of your systems? Tools and techniques for remote system management and monitoring will be discussed.

Deal with Tarballs

Not all software is in RPM packages. In fact a vast majority of them are delivered, by the developer, as a compressed tar file. When installed on an RPM based system, it's not visible to the system's RPM database--where the resulting loss of visibility can create issues. Methods and techniques on dealing with tarballs will be discussed.

Plan for Installation

Once a decision is made to install new software, how do you scope out the "net list" of tasks necessary to do the upgrade on all effected systems?

About Allan Cantos

Allan Cantos, Chief Technology Officer for Acrylis Inc. has been in software development for 18 years, managing and developing systems software and software development tools. Before joining Acrylis, Allan managed the engineering organization of UniPrise Systems, Inc. Prior to that, Allan managed a prominent software development consulting operation. Earlier in his IT career, Allan was the manager of Integration Platforms at Apollo, leading the design and development of software frameworks for tool integration. Allan has focused on the design and development of large heterogeneous systems tools in the UNIX and NT markets. Allan has an MS in Computer Science from the University of California at Berkeley.

About Acrylis Inc.

Founded in 1998 and headquartered in North Chelmsford Massachusetts, Acrylis Inc., is an emerging leader in the open-source, software management industry. Acrylis provides system administrators with Internet-delivered tools and services for faster, more reliable software management. The company has developed WhatifLinux, an Internet based subscription service that proactively monitors and manages the software assets running on networks of dynamic, Linux servers. Acrylis takes a unique approach to delivering critical Linux software information. By focusing their efforts on deploying autonomous agents that work in conjunction with their Knowledge Base, Acrylis is creating a community of open-source software know-how, which is designed for quicker, more reliable open- source software management. For more information, visit www.WhatifLinux.com.

For Additional Information Please Contact:

Chris McCoin
McCoin & Smith Communications LLC
508-881-0095
chris@mccoinsmith.com

Knowing When To Say No

Allan Cantos
CTO, Acrylis, Inc.

Introduction

Linux acceptance in corporate enterprises has moved from being on one or two evaluation systems to being used on many systems. While the benefits of Linux have been well documented, the software management issues are another matter. Specifically, how does an administrator manage and maintain software on multiple Linux systems given the dynamic change in free/open source software development.

Research done by Acrylis, Inc. shows that administrators spend between 30 minutes to an hour a day keeping up with the changes in free/open source software. They spend an additional 30 minutes to one hour evaluating how this software applies to their systems, and accessing what needs to be done to implement what they've just researched.

This paper outlines some of the issues and possible solutions.

Understanding Recursive Dependencies

Not long ago, new software was added to UNIX and Linux systems using 'tarballs' or more specifically, tape archive record files. As we know, tar files simply read and write what a directory of files has, and places them in the relative place on the system. Putting software in the correct directory is a good start, but much additional time is needed to get software configured to work properly. This includes setting environment variables, editing scripts and resource files, creating symbolic links, and more. The whole process is further complicated when dependencies arise when running the software for the first time. The administrator discovers a shared library was missing, or needs to be upgraded.

To resolve the issue of properly configured software, Marc Ewing and Eric Troan developed the Red Hat Package Manager (RPM). RPM addresses some of the complexities in installing, upgrading, and removing software. Today, RPM has become a standard, used by many of the Linux distributions in addition to developers wanting to deliver software to users.

With very few exceptions, all software depends on installed components resident on a target system. These dependencies are outlined in the dependency tag of the RPM Spec file that the target system uses to compare to the system's RPM database. For very simple packages, this system works well, but when a dependency has its own dependencies, or when one or two of the installed components, it doesn't quite work well. Other problems arise when these dependencies conflict with each other, or need to be updated to fulfill their support role.

Problems with Recursive Dependencies and RPM

The main problem with RPM installation today is that it doesn't the installer whether the software package being installed requires software that has subsequent dependencies. Administrators running into this problem can get caught up in a *version tango*, manually going down each dependency to find out what needs to be installed, what needs to be updated, or what is in conflict and why. The result is time consumed in installing software that could be spent doing other work.

Common remedies employed by Administrators faced with recursive dependencies includes using a --nodeps install, or using tarballs to install software. While these are the quickest paths to getting software installed, it has a high risk of not working correctly. What's left is software invisible to the other software in the RPM database, giving rise to other software incompatibilities and other hidden *gotchas* later.

One of problems causing recursive dependency is that information isn't filled out properly or completely in the package description tags, yielding an incorrect analysis from the RPM query. This problem makes it difficult to do any package equivalency analysis. Furthermore if an administrator installs a SuSE package on a Red Hat system the problem might get worse. More detail is provided in the table below, which contains that dependency information for the same version of Apache web server, but delivered by two different vendors:

Red Hat: apache-1.3.9-8-i386 Dependencies	TurboLinux: apache-1.3.9-6-i386 Dependencies
/etc/mime.types	/etc/mime.types
/sbin/chkconfig	/sbin/chkconfig
/bin/sh	/bin/sh
/usr/bin/perl	/usr/bin/perl
/bin/mktemp	
/bin/rm	
mailcap	
grep	
textutils	
ld-linux.so.2	ld-linux.so.2
libc.so.6	libc.so.6
libcrypt.so.1	libcrypt.so.1
libdb.so.3	libdb.so.3
libdl.so.2	libdl.so.2
libm.so.6	libm.so.6

Source: www.whatiflinux.com

These are essentially the same package, but Red Hat's release has five dependencies not found in the TurboLinux release. Which one is correct? Can the TurboLinux package be installed on the Red Hat system, and visa versa?

Some solutions to this problem exist. Debian already solves this by including recursive dependency information in its package descriptions, which is something RPM may implement in the future. The Linux Standards Base is an effort to solve many of the interoperability problems with Linux, including installing RPMs across distributions.

Managing Software Across All Systems

The complexity of software management on Linux systems increases when it is extended to dozens of systems stretched out over a corporate enterprise. If a security alert on a particular piece of software arises, how can you quickly find it on all of your systems? And once found, how easy is it to update?

Complicating the problem is the necessity for other people in the organization to have some choice in their system setup, and require access (root or sudo) to the system. This creates the possibility of system differences that need to be reconciled and managed when necessary updates need to be done.

One unfortunate solution to the problem of managing multiple systems is to mandate system configuration, essentially making all machines the same. This makes it easy to manage one system, and then push the results to all other systems. The disadvantage of this method is that systems can no longer be specialized for tasks (the Web server and the Samba server for example). Common Practice involves reducing the software configuration to only necessary daemons for increased security and reliability of the system. It also doesn't provide the necessary flexibility relevant to today's rapidly changing environment.

The choice for administrators for managing software across many systems includes scripting and record keeping (either manual or with a database). RPM lends itself to scripting because of its command line interface. Graphical applications such as Gnorpm can also be used when run remotely via X. The ultimate solution would allow an administrator to view all software on a specific system, and to view specific software over all systems.

Dealing with Tarballs

As mentioned earlier in this paper,not all software is in RPM packages. In fact, a vast majority of software is delivered by the developer as a compressed tar file. When installed on an RPM based system, the software is invisible to the system's RPM database. This loss of visibility can create issues with future software installations. This is especially true when dealing with dozens of systems, not all of which are under the control of the administrator.

Aside from auditing systems on a file-by-file basis, the only option for administrators is to get the system under control as soon as possible. Administrators can rebuild tars into RPMs using rpmbuilder. This application allows administrators to get packages under control before installation.

Plan for Installation

Once a decision is made to install new software, it's important to understand how to scope out the "net list" of tasks necessary to do the upgrade on all effected systems. The process is presented below:

1. **Find the most up to date, or most stable package.**

The many sources for this information present a challenge to administrators. Various source include the mirror sites, rpmfind.net. Red Hat charges for a service that gives customers priority access to FTP servers.

2. **Research the package.**

Review if the software has any outstanding security alerts. Check Bugtraq, Security Focus, X-Force, to see any issues.

3. **Verify package against public keys for vendor.**

Verifying a package can't occur until the software is downloaded. Can your source of the binary be trusted?

4. **Are there dependencies?**

If so, repeat above process until completed.

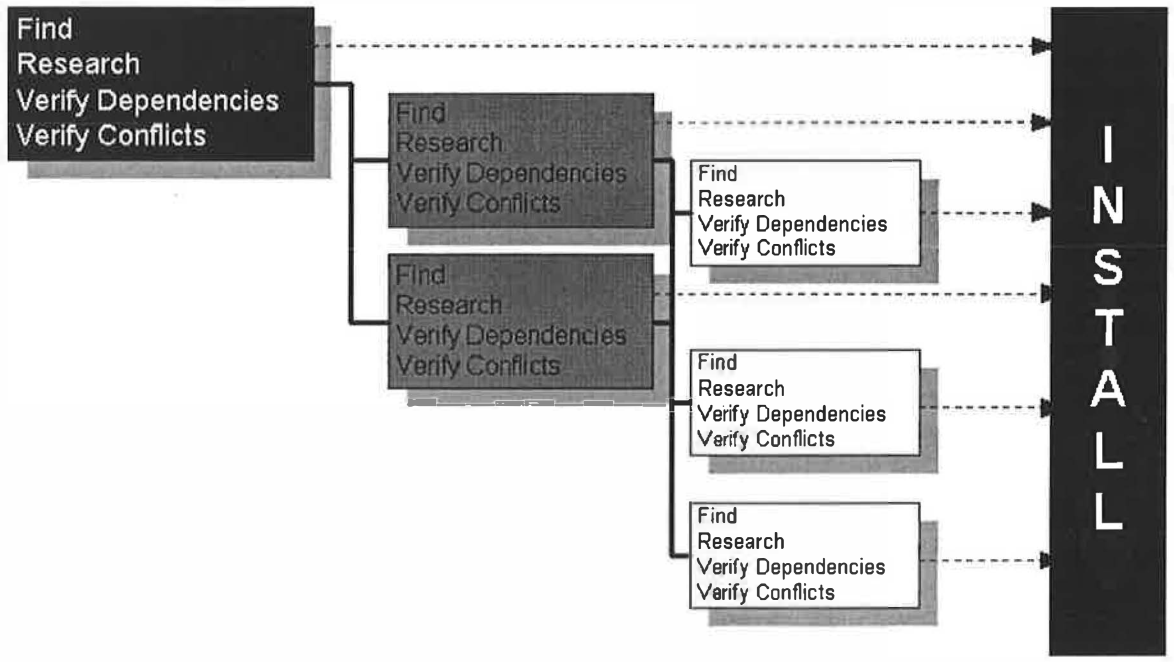
5. **Will installing a dependency impact other software running on the system?**

A risk assessment needs to be done whenever new libraries or other components are installed. Adding a new feature could possibly clobber an existing one.

If a problem is found (security or otherwise) during the installation process, the process needs to be repeated with the next available package. This is a very time consuming process, which lends itself to automation. Here's a graphical example of the process:

Conclusion

While current software installation and management systems have served us well up to this point, extending them to manage machines across the enterprise, with the level of dynamic change in the underlying software brings a whole new set of challenges. Undoubtedly, these challenges will be met by new methods in the near future.



Sources:

Maximum RPM, Edward C. Bailey, SAMS Publishing
copyright 1997by Red Hat Software Inc.
RPM Builder,
http://sourceforge.net/projects/rpmbuilder/
Linux Standards Base: http://linuxbase.org

Library Interface Versioning in *Solaris* and *Linux*

David J. Brown and Karl Runge

Solaris Engineering, Sun Microsystems Inc.

Abstract

Shared libraries in *Solaris* and *Linux* use a versioning technique which allows the link editor to record an application's dependency on a particular release level of the library. The versioning mechanism operates at the level of the library's GLOBAL symbol names—a finer granularity than simply associating a version number with the library itself.

In *Solaris*, this mechanism has also been used to provide a means for the system-supplied shared libraries to define their application interface: to declare specifically which of their symbols are intended for application use (and are stable from one release to the next), and which are internal to the system's implementation (and hence subject to incompatible change).

This paper describes the library symbol-versioning technology in *Linux* and *Solaris*, the ways in which it is used to support upward compatibility for existing compiled applications from one release of *Solaris* to the next, and the potential for similar mechanisms to be applied in *Linux* versioned shared libraries.

1 Introduction

As *Linux* continues to grow in popularity, and more people come to depend on it, issues regarding compatibility will likely become increasingly important. In what follows we focus on a method for successive releases of system software, such as *Linux*[1] or *Solaris,* to maintain binary compatibility with existing applications. We describe some practices that are presently being used in *Solaris* (at the level of the system's library interfaces) to define and maintain an Application Binary Interface (ABI). We further describe how this definition along with an approach to library interface versioning helps to ensure the *stability* of existing application binaries across successive *Solaris* releases.

The hope is that some, if not all, of these practices can be incorporated into the development practices for *Linux* libraries to help increase binary stability for *Linux* applications. Due to the differences in development models between GNU/*Linux* and *Solaris*, some aspects of these schemes may need to be modified suitably to be beneficial to *Linux* development.

2 The ABI—a basis for system interface definition

Maintaining source-level or API compatibility, is well understood; less familiar is the idea of maintaining binary-level, or ABI, compatibility. The *Solaris* ABI (application binary interface) is the set of *runtime* interfaces in *Solaris* that may be depended upon by an application; if the ABI evolves in an upward-compatible way from one release of *Solaris* to the next, then existing compiled applications built on a given release will run on all subsequent releases without change (i.e. there is no fear that the application will break when run on a later release of the system).

The enormous value of access to source code may obscure the fact that there are many end-users who cannot exploit source-level compatibility and "simply" recompile their applications as needed when the system software has changed. Most often it is large organizations, as compared to individual users or developers, who are in this situation. The need for the new system's binary compatibility with their existing applications *may* be because they do not have access to the source code for certain applications, but much more often the logistical nightmare of recompiling, retesting and redistributing the hundreds or even thousands of applications that a typical large organization relies on, is the predominant issue.

Consequently, establishing a clear runtime interface for applications, and maintaining its binary compatibility is

1. When we use the term *Linux* in this paper, we are using the common shorthand, and mean a complete software system based on a Linux kernel (those provided by RedHat, SuSE, or Debian GNU/Linux are examples).

an important requirement for a supplier of operating system software. The means of satisfying that requirement is through precise definition of the system's application binary interface (ABI): If the producer of system software maintains the integrity of those interfaces, and the developers of software products which rely on the system also abide by it (and only use interfaces belonging to the ABI) then compatibility is assured.

2.1 Defining the *Solaris* ABI

At Sun, we tend to think of the system (*Solaris*) as the provider of a set of services, and these services are primarily provided to application programs and/or other layered software products, which are built and released independently to *Solaris*.

To a first approximation, the broader set of system services are provided to applications by the set of libraries supplied as part of the system. While the kernel is the underlying foundation for this, as provider of the most fundamental services (and there can be a great deal of focus on this, especially in the *Linux* environment), a complete software system represents a good deal more than the kernel alone: So in *Solaris* it is the set of system libraries, and more specifically the particular interfaces offered by those libraries, that we use to characterize the application's runtime interface.

Importantly, these interfaces are accessed by a *compiled* software product (such as an application) via *dynamic* linking—bindings made at *runtime* between the application and the shared objects which implement the system libraries. The use of shared libraries is critically important because dynamic linking to these libraries allows us to maintain a clear separation between a compiled application and the system implementation. Focusing on the runtime binding interface between the two is analogous to defining a protocol, or may be thought of as characterizing the interface to the *Solaris* virtual machine.

2.2 Defining system-internal interfaces

While the system libraries provide a set of interfaces to allow applications access to system services, most system libraries also expose some implementation interface—making visible a broader set of GLOBAL symbols than just those intended for use by applications. System libraries which are "lower" in the system's implementation-architecture hierarchy can have intimate interdependencies, so these libraries also expose some system-internal implementation interfaces. These interfaces, such as those needed in `libc` to support particular semantics in conjunction with `libthread` or `libnsl`, are really part of the system's *implementation* (as opposed to its external interface), and not intended for application use. To distinguish these two classes of interface, we define the following terms:

"Public" - interfaces which *are* intended for use by applications (and/or any other layered software products released asynchronously to *Solaris*), and which therefore have the upward compatible evolution property;

and "Private" - interfaces which are *not* intended for use by applications (or any asynchronously released layered software component), because these interfaces are part of the internal implementation of *Solaris*, and do not have the upward compatible evolution property.

In the following sections, where we discuss how change is managed, we will be focusing on changes to the Public interface—namely, those system interfaces that affect applications (or any other layered software product released *asynchronously* to the system software). In *Solaris* we have not yet attempted to provide for the asynchronous release of individual system libraries, such as might be addressed by carefully managing change to the interface between system libraries (i.e. the Private, or internal implementation interface manifest by the libraries). The assumption at present, therefore, is that a release of the system constitutes (and requires) a *synchronous* release of all the system libraries along with the kernel.

3 Versioning

As a system evolves, whether by the addition of new functionality, or via changes to the system's implementation of existing functionality (as are frequently done to improve its quality or performance), there is the need to indicate the kind of change. This is important, because applications, and other software products have been constructed which depend upon this functionality. Since changes to the system's interfaces can affect existing applications, some means to indicate the nature of the changes made is highly desirable.

A particular property that we consider important, and that we're trying to realize, is that developers of applications (and many other layered software products) can be insulated from changes made to the system they rely on for many of the kinds of change to that system's software.

3.1 Kinds of change—"major", "minor" and "micro" releases

At the outset is important to define the kinds of change that can be introduced to a system software product (and which also apply at a finer granularity to individual components of a system software product). In this paper we will focus specifically on the system's libraries, since libraries are the primary components of the system software which provide interfaces to other software products built to run on that system. The following three terms define a simple taxonomy of change which is applicable to all systems:

A *major* release is an *incompatible* change to the system software, and implies that [some] applications dependent on the earlier major release (specifically those that relied upon the specific features that have changed incompatibly) will need to be changed in order to work on the new major release.

A *minor* release of the system software is an *upward-compatible* change—one which adds some new interfaces, but maintains compatibility for all existing interfaces. Applications (or other software products) dependent on an earlier minor release will *not* need to be changed in order to work on the new minor release: Since the later release contains all the earlier interfaces, the change(s) imparted to the system does not affect those applications.

A *micro* release is a compatible change which does *not* add any new interfaces: A change is made to the *implementation* (such as to improve performance, scalability or some other qualitative property) but provides an interface *equivalent* to all other micro revisions at the same minor level. Again, dependent applications (or other software products) will *not* need to be changed in order to work on that release as the change imparted to the system (or library) does not undermine their dependencies.

3.2 Managing changes to the system interface

Now let's take a look at how the various kinds of change to system interfaces described above have been managed in some systems historically.

Virtually all systems that we are aware of incorporate some sort of version number in the filename of each library, and most systems that we are aware of have had some means of recording both the major and minor release concepts on a per-library basis in order to indicate and manage upward compatibility. Historically, several systems have recorded minor and/or micro release levels explicitly in the library's filename: For example, in Sequent's *Dynix* [Huiz 97] and Sun's *SunOS 4.x* (*Solaris's* OS component prior to *Solaris2*), library filenames were of the sort:

`lib<name>.so.<major>.<minor>`

(for example, `libc.so.2.9`), and in *Linux*, library filenames often also contain a micro (or "release") number, being of the sort:

`lib<name>.so.<major>.<minor>.<rls>`

(for example, `libz.1.1.3`).

3.2.1 Library minor release

When a library evolves compatibly, existing interfaces are preserved, but new ones are added (as needed for example, when new functional content is added). On many systems this change is reflected in the library (or libraries) affected by exhibiting both major and minor version numbers in the library's filename (e.g., `libfoo.so.1.2`) and incrementing the library's minor number to indicate that new interfaces were added (e.g. to `libfoo.so.1.3`). Since nothing has been done that would break applications constructed earlier (i.e. those that were built using either `libfoo.so.1.1` or `libfoo.so.1.2`), it is OK for these older applications to be linked with the newer library at run-time (although we have yet to describe how the application's dependencies are recorded and how this is managed at runtime, but will do so shortly).

3.2.2 Library major release

If the interfaces in a library shared object change incompatibly, not only must there be a way of indicating that a different version of the library has been created, but also of detecting application binaries that were built using the earlier edition of the library and preventing them from linking at runtime with the library's new major release.

Analogous to a library's minor release, the system or library provider's practice is to update the major revision number associated with the library. So for example, an incompatible change to `libfoo` would require that the successor to `libfoo.so.1` be named `libfoo.so.2`, or in the case that both major and minor numbers are reflected in the library filename (as in our preceding example), the naming revision would, for example, be from `libfoo.so.1.3` to `libfoo.so.2.1` (where 2.1 indicates the first minor release of major release 2).

3.3 How applications record library dependencies

Now while updating the major and/or minor version number on a library is a simple way for the system software or library provider to indicate the change, this practice presumes of course, that there is also some mechanism for labelling application binaries with the revision levels of the libraries they have used. And at run-time, some mechanism must be present to ensure a rendezvous of the application with an appropriate version of each library it requires.

There must, at the very least, be a way of marking applications with the name and *major* revision level of a library they were built with to ensure that application executables requiring a given major version of a library (e.g. `libfoo.so.2`) are not accidently linked with another major version of the library, either a later (`libfoo.so.3`) or earlier one (`libfoo.so.1`), since we know that these are incompatible with the application. Related to this is that later releases of a system software product, if they are to be able to run older application binaries, must continue to provide earlier major edition(s) of the libraries, and have a mechanism for the older applications to be linked at runtime to the major version they require.

3.3.1 Historical Practice

On some earlier systems, when an application was built, the link editor simply recorded within the application binary, the *filename* of each library that it depended upon (that is, a name containing the major and minor number of the library present on the build platform).

At run time, upward compatibility could be handled by the runtime linker's explicit knowledge of the semantics of the version numbers contained in the library filenames. In SunOS 4.x for example, the runtime linker would look for a library with the same name and same *major* release number as that recorded as a dependency in the application (and in the presence of multiple minor-version instances of the library that match the major number, the instance with the highest minor version number is used) [Gingell 87].

If the minor version number of the library found on the runtime platform was greater or equal to that of the dependency recorded in the application binary, dynamic linking proceeded silently (since the library must contain all the content required). In the case that the minor revision found on the runtime platform was lower than that recorded in the binary SunOS 4.x had the policy of issuing a warning diagnostic (that an earlier version had been found), but allowing the runtime linking to proceed. The application *might* still run successfully if it had only happened to depend on functional content (and interfaces) present in the earlier release of the library. If the application had depended on later content however, a runtime relocation error would occur when the application invoked an interface not present in the library on the runtime platform. Other systems may have adopted a more conservative policy and disallowed an application to run if a minor release of at least equal minor revision level to that required by the application was present on the runtime platform.

Neither of these policies is particularly satisfactory however, as the former might permit an application to run whose dependencies can't be met, while the latter prevents a class of applications whose interface dependencies could be met (i.e. whose interface needs were limited to content present in an earlier minor edition of the library, but were built on platforms with later minor editions), from being allowed to run on platforms bearing an earlier edition of the library than the application was built (and hence labelled) with, but which in fact happened to provide everything the application actually required.

3.4 The minor-version rendezvous problem

The important but somewhat subtle issue associated with *minor* revision changes, just described, is that an application built with a given minor release of a library *might*, but *cannot be certain* to run on an earlier minor release level of the library. This is because the application may have used one or more of the interfaces added to the library at a later minor release level, and not present in the earlier one.

To resolve this problem we must know more than just what minor release level an application was built with. We must know more specifically what content within that library it is actually *dependent upon* if we are to determine whether that content is present in the library found on a particular runtime system that the application is being run against. This is one of the key concerns addressed by the library versioning and linking technology present in both *Solaris* and *Linux*-based systems[2].

4 Library versioning in *Solaris* and *Linux*

Many contemporary systems (including *Solaris* and *Linux*-based systems) use the ELF object file format (the SystemV Executable and Linking Format) [SysVABI]. Where ELF is used, dynamically-linked libraries (libraries implemented as shared objects) contain an *so-name*—a specific means of naming the library (superceding the library's filename) stored within the library's object file[3].

When an application (or other dynamic object) which depends on the library is built, it is the library's *so-name* (not the filename) that is recorded in the application binary as a dependency.[4] And when the application is

2. Those using GNU libc version 2 or later
3. The DT_SONAME contained in the shared object's .dynamic section.

run, dependency information contained in the application binary is used by the runtime linker to locate and load the libraries depended upon by the application.

In order to allow for upward compatible evolution of the library—to permit an application constructed on a given system release to encounter a different minor revision of the library on the runtime platform (and still run successfully), current practice is to have the library's *so-name* contain only the major number (e.g. `libc.so.1`). At this level applications record a dependency only on the particular name and *major* release of the library and may be run with any minor release level they encounter on a runtime platform (albeit with the expectation of finding a minor release level sufficient to provide the interface content they depend upon).

For minor versioning, rather than the traditional method of associating a single minor version number with the library and incrementing it (as a way of saying "something was added"), a more useful approach is to define specifically *what* has been added, and to record this information in the library shared object itself.

So, instead of simply renaming the library (e.g. from `libfoo.so.1.2` to `libfoo.so.1.3`), the library's name remains `libfoo.so.1` (reflecting its major release level) and inside the library, a label is introduced (say, "VERS_1.3") that indicated the GLOBAL symbols added in the third minor revision level. If such labels are added with each minor revision level (e.g. VERS_1.1, VERS_1.2, VERS_1.3, ...) and all earlier labels preserved within subsequent minor release editions of the library, the evolution of the library's interface can be seen clearly.

4.1 Basic mechanism

In 1995 the *Solaris* 2.5 link editor (`ld`) and the run-time linker (`ld.so.1`) were enhanced to support "versioning" and "scoping" of symbols in shared objects.

The versioning mechanism itself is fairly stratightforward, simply allowing for the definition of named sets, each of which contains a specified list of symbols. Sets may be defined by providing an explicit list of symbols, and/or by referring to other sets by their name to include (inherit) the symbols in those sets[5] [Solaris LLM].

4. More specifically the *so-name* is recorded as a DT_NEEDED entry in the application object's .dynamic section for each library used by the application.
5. Full details of the library versioning technology in Solaris may be found in the *Solaris Linker and Libraries Manual* in the section describing "Versioning".

The GNU/*Linux* implementation[6] is the same as that in *Solaris*, although the GNU implementation provides two extensions [GNU_ld]: First, as an alternative to providing definitions in a separate mapfile, "`.symver`" assembler directives may be included in-line in the C source code for the library. Second, a form of interface overloading is provided: Multiple (incompatible) versions of the same function are allowed to exist in a single revision of the library. This is done by mapping an external symbol name (as referred to by an application) to a different internal name for the function, depending on the (minor) version set specified by the application's dependency. Special `.symver` directives must be provided to indicate the per-version mapping:

```
.symver old_printf, printf@VERS_1.1
.symver new_printf, printf@VERS_2.0
```

The intention is to allow several incompatible versions of any individual interface to be carried simultaneously within the library, and thus not have to increase the library's major version number[7].

When a *versioned* library (shared object) is built, a "mapfile[8]" containing the list of exported symbols grouped into named sets[9] is passed to the link editor `ld(1)` via the `-M` option.

As a simple example, consider the versioning mapfile for a hypothetical library named `libstack.so.1`:

```
SUNW_1.1 {
    global:
        pop;
        push;
}

SUNWprivate {
    global:
        __pop;
        __push;
    local:
        *;
}
```

6. The mechanics for symbol versioning in *Linux* are implemented in the GNU link editor (`ld`), provided in the `binutils` package
7. We are not yet sure of the broader policies describing the expected use of this mechanism however.
8. What is called a "versioning mapfile" in Solaris is called a "VERSION script" by the GNU link editor, but their syntax is the same.
9. These named sets are called "versions" in the parlance of the Solaris Linker and Libraries Manual, indicating the primary purpose intended by their design.

In *Solaris* system libraries, by convention, the "SUNW" prefix[10] is used in the set names (versions).

The versioning mapfile above instructs the link editor (`ld(1)`) to construct a shared object which exports (as GLOBALs) the symbols `pop`, `push`, `__pop`, and `__push` for use by other binary objects (executables or other shared objects). The "`local: *;`" directive instructs the link editor to take all remaining GLOBAL symbols defined in the objects being linked and make them inaccessible external to the shared object being produced: Essentially the link editor "demotes" these symbols from GLOBAL to LOCAL symbols (see 4.1.2 "Scoping" below). For example, utility functions that are part of the internal implementation interface of the library (and hence intended only for use within the shared object) will not be exported.

Suppose that in a later revision of `libstack.so.1` it is decided that a `swap()` functionality is desired, then the mapfile would be the same as above, but with an additional version definition:

```
SUNW_1.2 {
    global:
        swap;
} SUNW_1.1;
...
```

This notation reflects the upward-compatible evolution of the library's Public interface, in which the set SUNW_1.2 defines two new interfaces, and inherits those interfaces in the set named SUNW_1.1. The inheritance chain of symbol sets SUNW_1.1 .. SUNW_1.2 ..., and so on, evolves corresponding to each new revision that adds interfaces to the library. Note that the version numbering scheme following the SUNW prefix is a major and minor number pair, where the major number corresponds to the major revision number of the library.

4.1.1 Versioning

The immediate effect of the library's versioning is that at the time an application is built (compiled and linked), the link editor can record into the application binary the names of any versions (named sets of symbols) in the library that the application depends on. This is the default build practice for *Solaris* applications. Importantly, it is not the name of the *latest* set (version) present in the library that is recorded, but the smallest set (or sets) containing those symbols *depended upon* by the application: For example, if `libc.so.1` contained six minor revision levels, of which the latest was `SUNW_1.6` on the platform used to build `test_app`, but this application only relied on symbols present in revisions up to the third minor release (`SUNW_1.3`), then the application would be labelled with that named set to indicate its correct minor version dependency.

This permits applications built on later minor release editions of the library to be run (correctly) with earlier editions of the library, when their interface requirements are constrained to an earlier release level. And second, it ensures that applications which record a dependency on a given named set (minor revision level) will not be run[11] with an edition of the library which does not possess that named set.

When the application is run, the runtime linker uses the version dependency information recorded in the application binary to determine if all these named sets (interfaces required by the application), are present in the library found on the runtime platform. This ensures that the sufficient *minor* revision content is present in the library to meet the application's needs (thus going beyond simply using the application's list of NEEDED *sonames* to locate the correct *major* versions of the libraries).

4.1.2 Scoping

Somewhat specifically designed to overcome a shortcoming of the C language's symbol scoping capabilities, implementation interface which is used *only* internal to a library itself (i.e. interface used only within a single dynamic object) can be handled specially. A capability is afforded by the *Solaris* link editor which permits a reduction in the scope of those interfaces from GLOBAL to LOCAL within the library at the time that the library (dynamic object) is linked. We refer to this as "scope reduction" or library-level "scoping" of symbols.

The keyword `local:` in a mapfile is a scope-reduction directive, and provides that one or more symbols intended for use only within the shared object itself may be treated as library-level STATIC symbols. In this way the shared library can control what symbols are intended for export. Scope-reduced symbols are changed from library GLOBAL symbols to LOCAL symbols as part of the

10. Originally this was intended as a way to distinguish interfaces introduced by Sun Microsystems—and therefore perhaps particular to Solaris, from those defined by broader standards, such as the System V ABI or the Open Group's UNIX). The prefix "GLIBC" is used by the GNU/glibc package in a similar convention.

11. At application start-up, a warning is emitted by the Solaris runtime linker indicating that the library does not contain the version required by the application, and the application exits. This is in lieu of a runtime relocation error if the application were allowed to execute.

link editing process which produces the shared object, thus preventing application programs (or any other dynamic object) from accidentally (or intentionally) using them. As a corresponding effect, these symbols are removed from the dynamic symbol table. Note that scope-reduced symbols are not actually associated with any named set.

4.2 Versioning practices (policies) in *Solaris*

To implement *Solaris's* interface definition and upward compatibility policies, we have defined a set of practices which apply the library versioning mechanisms described above. These practices are now used at Sun as an intrinsic part of *Solaris*'s library development practices.

Sun defines the *Solaris* ABI in terms of the interfaces to the system's libraries. First, at the library-naming level, libraries are given a filename and *so-name* corresponding to the library's major release level. Minor versioning information is contained within the library binary.

In order to make clear which of a library's GLOBAL symbols are part of the ABI and which aren't, the symbol versioning mechanism described above are used to classify all GLOBAL symbols (as Public or Private, and by ascribing each Public interface to a set indicating the minor release level of the library in which it was introduced). The set of all symbols indicated to be Public in a system library constitutes the library's ABI, and the collection of all such libraries in a given release of *Solaris* thus constitutes the *Solaris* ABI. This ABI is self-documenting since the definitional information (which symbols are Public and which are Private) is part of the library itself[12] and readily accessible through system utilities such as the `pvs(1)` ("print version section") command.

When the library versioning mechanisms were first applied to the *Solaris* shared libraries, the scoping mechanism was applied to hide all interfaces that are part of the linkage between the individual compilation units (`.o` files) of the library, but used *only* within the library itself. These symbols are "scoped out" (demoted from GLOBAL to LOCAL in the link-editor's construction of the shared object), so that these symbols are not visible external to the library and cannot be used by any external dynamic object.

Remaining GLOBAL symbols (those interfaces that must be visible external to the library) are separated into Public and Private. GLOBAL symbols classified as Public name interfaces intended for use by application developers (they are documented and guaranteed not to change incompatibly from one release of *Solaris* to the next). Private symbols name interfaces that are part of the *Solaris* implementation (they can not be guaranteed to remain compatible, or even to persist at all, from one Solaris release to the next, and are not suitable for use by application developers).

To reflect the upward compatible evolution represented by a series of minor revisions to the library, the Public symbols appear as a number of named sets of the form: "SUNW_*<major>*.*<minor>*". Each named set (version) identifies the full interface content present in a given minor revision of the library. The set lists the Public symbols introduced in that minor release, and names its predecessor to inherit its contents (e.g. SUNW_1.2 explicitly identifies the set of symbols added in the second minor release of `libc`, and inherits SUNW_1.1—the set of symbols present prior to that). A new version is added to the library only when a release of the library introduces new interface content.

All Private symbols, in contrast, are associated with a single version named "SUNWprivate". Symbols may be added to (or removed from) this set from one release to the next, and since there is no expectation of upward compatibility in this set there is no inheritance chain of versions for Private symbols. Recall that Private means (system-internal implementation interface) and that applications must not depend on these symbols. The contents (or even the existence) of this set therefore should not matter to an application.

All of the system libraries in *Solaris* which provide the basic OS and core networking services, as well as many of the basic window system interfaces, have been versioned in this way since *Solaris 2.6*. The eventual goal is to version all libraries shipped by Sun which can be used with *Solaris*. In due course it is hoped that the same approach will be taken by libraries built by other developers—particularly those "middleware" products which are not included with the *Solaris* release, where such libraries offer application-usable interfaces. The intent is that all layered products that can be used with

12. These definitions are contained within each library binary. They are reflected within the shared object by three ELF sections: Two sections named .SUNW_version and one named .SUNW_versym. The first has sh_type: SHT_SUNW_verdef, and gives all those versions (named sets of symbols) defined by the library. The second section has sh_type: SHT_SUNW_verneed and lists versions (named sets of symbols in other shared objects) depended upon by the library). The third has sh_type: SHT_SUNW_versym, and associates a set of GLOBAL symbols in the library with a respective "version" (named set) listed in the first section in order to define each such set name.

Solaris define *stable* application interfaces, in order to realize similar benefits of upward binary compatibility for applications that depend upon them.

4.3 Versioning practices in *Linux*-based systems

The GNU "glibc" package provides about 20 shared libraries (including `libc`) and makes extensive use of the versioning mechanism in `ld(1)`, both to implement scope reduction for library-internal symbols, and to indicate the library's minor release evolution through versioning. For example, in `libc.so` the current version chain is:

GLIBC_2.0, GLIBC_2.1, GLIBC_2.1.1, GLIBC_2.1.2

For libraries that have not been added to recently, the highest version remains the last one in which content was added. For example, the highest version in `libcrypt.so` is GLIBC_2.0. If a new library is added at a certain version of the GNU glibc package its initial version set name is that of the corresponding package: e.g. `librt.so` begins at GLIBC_2.1

Looking at the Redhat Linux 6.2 release, one can see that most of the libraries that are not part of the glibc package (e.g. those of `XFree86` and `libgtk`) are not as well managed: While most have versions defined in them, these are currently only a default version with no structure yet defined (that is, there are only two versions `lib<`*name*`>.so.<`*n*`>` and `GCC.INTERNAL`). These libraries currently have no Public inheritance set chain defined.

The most important difference between GNU/*Linux* and *Solaris* is that the GNU `glibc` libraries do not distinguish the system's internal implementation-interface[13] from their application interface. By making it clear that application developers should not use implementation interfaces (see section 5), *Solaris* library developers can change the library's *implementation* in the future (for example, to substitute new algorithms or to achieve performance gains within the Solaris system libraries), without the fear that existing applications could be broken.

5 Constructing stable applications

Once a system has clearly defined the set of runtime interfaces intended for use by applications[14], and is committed to maintain them in an upward compatible way, all *properly constructed* applications will continue to run without change. This raises the question of how we decide that any given application meets those criteria.

5.1 `appcert`: Checking applications' interface use

An immediate benefit of the Solaris ABI is that we can use the definition it provides to decide whether a compiled application (or other software product) uses unstable interfaces. This can be done by a tool which:

- Determines all bindings an application makes to interfaces in Solaris's libraries.
- Extracts the system's interface definition information (Public vs. Private interfaces) from the Solaris libraries.
- Warns of any bindings made directly from the application to Private (non-ABI) interfaces in the libraries.

We have written a tool for *Solaris* that performs the above examination and one or two other checks for potential binary instabilities (for more information see [appcert]).

Implemented as a Perl script, `appcert` relies on two important *Solaris* system utilities: To determine an application's runtime dependencies (both the libraries and specific per-library symbol bindings) `appcert` relies on a feature of the *Solaris* runtime linker (`ldd(1)`)[15]. Next, for each *Solaris* system library the application depends on, `appcert` uses the `pvs(1)` utility to determine the library's ABI (its Public vs. Private symbols).

Some additional checks related to binary stability are also performed by `appcert`. In particular, the static linking of *Solaris* archive libraries (e.g., `libsocket.a`) are flagged, as well as calls to certain specific interfaces—whether individual symbols or entire libraries, known to have caused binary breakage in earlier releases.

13. Some analog to the SUNWprivate symbol set that *Solaris* system libraries use to indicate unstable inter-library artifact which is not part of the ABI as opposed to stable interfaces that application developers are intended to use.

14. This also applies to any other layered software product that is not part of the system (in the sense that it does not such an integral part of the core system software that it must be re-built and reissued as a part of every release of the system software product).

15. ldd is run with the environment variable LD_DEBUG set to "files,bindings".

6 Benefits

Library versioning, as present in both *Solaris* and *Linux* provides a finer grain solution to the minor-revision rendezvous problem described above. An application which has been constructed using versioned libraries records the name(s) of the version(s) containing the interfaces that it uses, and that it thus requires to be present in a library on a runtime platform. Beyond location of a library matching the major revision level needed by the application on the runtime platform (an exact match of the *so-name* recorded in the application binary), the runtime linker now also ensures that the minor version dependencies recorded in the application are present within the library.

Library scoping has been applied to eliminate a class of library internal interface from external visibility. Dynamic linking of the libraries is sped up by scoping, since scoped symbols are removed from the dynamic symbol table (.dynsym): Since scope-reduced symbols become LOCAL symbols, references to those symbols (within the library) are resolved statically at the time the library is constructed. Dynamic relocations are no longer required for these symbols.

In *Solaris*, library versioning has also been applied to define the ABI—a stable, upward compatibly evolving interface for applications, and to distinguish this from a set of interfaces exposed by libraries which reflect part of the system's internal implementation. This serves as the foundation for ensuring the integrity of successive system releases, and for establishing stability in the installed base of applications and software products that rely on the system.

7 Conclusions

Given that the enabling technology is now present in the GNU linker, and has been demonstrated in its application to `glibc`, it strikes us as highly desirable that additional libraries used by *Linux* developers (e.g. `XFree86` and `libgtk`) adopt versioning practices consistent with those used by the GNU `glibc` libraries. An important part of this will be to identify and advocate a set of *policies* to be used—especially important considering the number of independent developers contributing to *Linux*-based systems. The more libraries that carefully define and manage the evolution of their external interfaces, the smaller is the chance for binary incompatibilities to arise for applications that depend upon them. And the more *uniform* the set of practices for implementing library interface definitions, the more practical will be developers ability to understand and apply that in the software products that they construct.

7.1 A *Linux* ABI

We are convinced that the GNU `glibc` project (and other *Linux*-related library projects) would benefit if a GNU/*Linux* ABI were defined for these libraries. This could be done, just as in *Solaris*, by adding an analog to SUNWprivate (for example, a "GLIBC_PRIVATE" for the GNU `glibc` package), to indicate the system-internal (non-ABI) symbol set. Currently both application interface (ABI) and system-internal interface (non-ABI) symbols appear to be exported together.

If the *Linux* community discovers these practices to be effective, it should be as natural to define all library interfaces as it was for *Solaris*. In fact, due to the more distributed and modular nature of open source development, it may prove even more fruitful to apply these practices. Further, due to the independent development and release of many of the libraries used in *Linux*-based systems, it may be necessary to explore additional classifications beyond the "Public" and "Private" used in *Solaris*, perhaps to identify and version inter-library interfaces (those between separately-released collections of libraries). While it may appear that this use of symbol versioning only applies to monolithic "cathedral" systems like *Solaris*, it should be noted that Sun also applies its versioning scheme to libraries from outside Sun (e.g., the CDE and X11 libraries) released as part of *Solaris*.

Given the similarity of mechanisms, the definition and use of the ABI to cultivate a base of increasingly stable applications, as in *Solaris,* could easily be done in *Linux*. As an initial step, and proof of concept, we have developed a prototype of `appcert` on *Linux*. But while the tool itself is a necessary element of the solution, the identification of a core set of system libraries for *Linux* systems and the definition and stabilization of their interfaces is needed to realize the full value. Such considerations might serve as the basis for a broader discussion of what libraries constitute the core system interface for *Linux*, and what interface definition and versioning practices might be useful to the open source community and development process.

Our ultimate desire is that a set of normalized practices for library interface definition and management of compatibility will be identified that are sufficient for common and widespread use in the industry.

7.2 Compatibility across *Linux* systems

Sun has benefitted from the library versioning practices-described, both by defining *Solaris's* system interfaces and in managing their upward compatible evolution. We are excited to see these mechanisms and similar practices adopted by the GNU `glibc` project, and hope that the practices will be developed and applied more broadly in open source library development.

A significant opportunity that arises from the definition of an ABI and library versioning, is the ability to compare the system interface provided by different system or product releases. While most recently this has been used in *Solaris* to maintain upward compatibility for successive releases of a *single* product, definition of an ABI in the *Linux* environment could serve to enable *cross-product* binary compatibility, so that a software product build on one *Linux*-based system (such as a Caldera release) could be run successfully on others (such as *Linux*-based distributions released by RedHat, Debian, SuSE and so on). This could prove important to avoid a Balkanization of the interface as offered by different *Linux*-based releases, and perhaps critical to the success of the open source efforts related to its ongoing development.

References

[appcert] "Solaris appcert tool", available at URL: http://www.sun.com/developers/tools/appcert.

[Gingell 87] Robert A. Gingell, Meng Lee, Xuong T. Dang and Mary S. Weeks, "Shared Libraries in SunOS", USENIX Conference Proceedings, pp. 131-147, Summer 1987, Phoenix, AZ.

[GNU_ld] Info pages for the GNU linker (ld). File:ld.info, Node:VERSION, GNU glibc version 2.x.

[Huiz 97] Gerritt Huizenga, Dynix library versioning practices, personal communication, 1997.

[Johnson 98] Michael K. Johnson and Eric W. Troan, "Linux Application Development", Addison Wesley Longman Inc., (c) 1998, Reading, MA, ISBN: 0-201-30821-5.

[Solaris LLM] "Solaris Linker and Libraries Manual", available at URL: http://docs.sun.com/ab2/coll.45.13

[SysVABI] "System V Application Binary Interface", ISBN 0-13-100439-5, UNIX Press (Prentice Hall), Englewood Cliffs, N.J. (c) 1993.

JFS Log

How the Journaled File System performs logging

Steve Best sbest@us.ibm.com
IBM Linux Technology Center

Abstract

This paper describes the logging done by the Journaled File System (JFS). By logging, JFS can restore the file system to a consistent state in a matter of seconds, versus minutes or hours with non-journaled file systems. This white paper gives an overview of the changes to meta-data structures that JFS logs.

Introduction

The Journaled File System (JFS) provides a log-based, byte-level file system that was developed for transaction-oriented, high performance systems. Scaleable and robust, one of the key features of the file system is logging. JFS, a recoverable file system, ensures that if the system fails during power outage, or system crash, no file system transactions will be left in an inconsistent state. The on-disk layout of JFS will remain consistent without the need to run fsck. JFS provides the extra level of stability with a small decrease in performance when meta-data changes need to be logged.

JFS uses a transaction-based logging technique to implement recoverability. This design ensures a full partition recovery within seconds for even large partition sizes. JFS limits its recovery process to the file system structures to ensure that at the very least the user will never lose a partition because of a corrupted file system. Note, that user data is not guaranteed to be fully updated if a system crash has occurred. JFS is not designed to log user data and therefore is able to keep all file operations to an optimal level of performance.

A general architecture and design overview of JFS is presented in [JFSOverview].

The development of a recoverable file system can be viewed as the next step in file system technology.

Recoverable File Systems

A recoverable file system, such as IBM's Journaled File System-(JFS), ensures partition consistency by using database-logging techniques originally developed for transaction processing. If the operating system crashes, JFS restores consistency by executing the logredo procedure that accesses information that has been stored in the JFS log file.

JFS incurs some performance costs for the reliability it provides. Every transaction that alters the on-disk layout structures requires that one record be written to the log file for each transaction's sub-operations. For each file operation that requires meta-data changes, JFS starts the logging transaction by calling the TxBegin routine. JFS ends the transaction's sub-operation by calling the TxEnd routine. Both TxBegin and TxEnd will be discussed in more detail in a later section.

Logging

JFS provides file system recoverability by the method of transaction processing technique called **logging**. The sub-operations of any transactions that change meta-data are recorded in a log file before they are committed to the disk. By using this technique, if the system crashes, partially completed transactions can be undone when the system is rebooted. A transaction is defined as an I/O operation that alters the on-disk layout structures of JFS. A completed list of operations that are logged by JFS will be discussed later. One example of an operation that JFS logs is the unlink of a file.

There are two main components of the JFS logging system: the **log file** itself and the **transaction manager**. The log file is a system file created by the mkfs.jfs format utility.

There are several JFS data structures that the transaction manager uses during logging and they will be defined next.

Extents, Inodes, Block Map

A "file" is allocated in sequences of extents. An **Extent** is a sequence of contiguous aggregate blocks allocated to a JFS object as a unit. An extent is wholly contained within a single aggregate (and therefore a single partition); however, large extents may span multiple allocation groups. An extent can range in size from 1 to 2(24) -1 aggregate blocks.

Every JFS object is represented by an inode. Inodes contain the expected object-specific information such as time stamps and file type (regular vs. directory, etc.).

The Block Allocation Map is used to track the allocated or freed disk blocks for an entire aggregate.

The Inode Allocation Map is a dynamic array of Inode Allocations Groups (IAGS). The IAG is the data for the Inode Allocation Map.

A more complete description of JFS' on-disk layout structures is presented in [JFSLayout].

Transaction Manager

The Transaction Manager provides the core functionality that JFS uses to do logging.

A brief explanation of the transaction flow follows:

A call to TxBegin allocates a transaction "block", tblk, which represents the entire transaction.

When meta-data pages are created, modified, or deleted, transaction "locks", tlck's, are allocated and attached to the tblk. There is a 1 to 1 correspondence between a tlck and a meta-data buffer (excepting that extra tlcks may be allocated if the original overflows). The buffers are marked 'nohomeok' to indicate that they shouldn't be written to disk yet.

In txCommit (), the tlck's are processed and log records are written (at least to the buffer). The affected inodes are "written" to the inode extent buffer (they are maintained in separate memory from the extent buffer). Then a commit record is written.

After the commit record has actually been written (I/O complete), the block map and inode map are updated as needed, and the tlck'ed meta-data pages are marked 'homeok'. Then the tlcks are released. Finally the tblk is released.

Example of creating a file

A brief explanation of the create transaction flow follows:

```
TxBegin(dip->i_ipmnt, &tid, 0);

tblk = &TxBlock[tid];

tblk->xflag |= COMMIT_CREATE;

tblk->ip = ip;

/* Work is done to create file */

rc = txCommit(tid, 2, &iplist[0], 0);

TxEnd(tid);
```

File System operations logged by JFS

The following list of file system operations changes meta-data of the file system so they must be logged.

- File creation (create)
- Linking (link)
- Making directory (mkdir)
- Making node (mknod)
- Removing file (unlink)
- Rename (rename)
- Removing directory (rmdir)
- Symbolic link (symlink)
- Set ACL (setacl)
- Writing File (write) (not on normal conditions)
- Truncating regular file

Log File maximum size

The format utility mkfs.jfs creates the log file size based on the partition size.

The default of the log file is .4 of the aggregate size and this value is rounded up to a megabyte boundary. The maximum size that the log file can be is 32M. The log file size is then converted into aggregate blocks.

For example, the size of the log file for 15G partition using the default is 8192 aggregate blocks using 4k-block size.

Logredo operations

Logredo is the JFS utility that replays the log file upon start-up of the file system. The job of logredo is to replay all of the transactions committed since the most recent synch point.

The log replay is accomplished in one pass over the log, reading backwards from log end to the first synch point record encountered. This means that the log entries are read and processed in Last-In-First-Out (LIFO) order. In other words, the records logged latest in time are the first records processed during log replay.

Inodes, index trees, and directory trees

Inodes, index tree structures, and directory tree structures are handled by processing committed redopage records, which have not been superseded by noredo records. This processing copies data from the log record into the appropriate disk extent page(s).

To ensure that only the last (in time) updates to any given disk page are applied during log replay, logredo maintains a record (union structure summary1/summary2), for each disk page which it has processed, of which portions have been updated by log records encountered.

Inode Allocation Map processing

The extent allocation descriptor B+ tree manager for the Inode Allocation Map is journaled, and a careful write is used to update it during commit processing. The inode map control page (dinomap_t) is only flushed to disk at the umount time. For iag_t, persistent allocation map will go to disk at commit time.

Other fields (working allocation map. sum map of map words w/ free inodes, extents map inode free list pointers, and inode extent free list pointers) are at working status (i.e. they are updated in run-time). So the following meta-data of the inode allocation map manager needs to be reconstructed at the logredo time:

- The persistent allocation map of inode allocation map manager and next array are contained in Inode Allocation Groups.
- Allocation Group Free inode list
- Allocation Group Free Inode Extent list
- Inode Allocation Group Free list
- Fileset imap

Block Allocation Map (persistent allocation map file) is for an aggregate. There are three fields related to the size of persistent allocation map (pmap) file.

1. superblock.s_size: This field indicates aggregate size. It tells number of sector-size blocks for this aggregate. The size of aggregate determines the size of its pmap file. Since the aggregate's superblock is updated using sync-write, superblock.s_size is trustable at logredo time.
2. dbmap_t.dn_mapsize: This field also indicates aggregate size. It tells the number of aggregate blocks in the aggregate. Without extendfs, this field should be equivalent to superblock.s_size. With extendfs, this field may not be updated before a system crash happens. So logredo could need to update it. Extendfs is the JFS utility that could provide the functionality to increase the file system size. Ideally, the file system should have its size increased by using the Logical Volume Manager (LVM).
3. dinode_t.di_size: For an inode of pmap file, this field indicates the logical size of the file. (I.e. it contains the offset value of the last byte written in the file plus one. So di_size will include the pmap control page, the disk allocation map descriptor control pages and dmap pages. In JFS, if a file is a sparse file, the logical size is different from its physical size.

Note: The di_size does not contain the logical structure of the file, i.e. the space allocated for the extent allocation descriptor B+ tree manager stuff is not indicated in di_size. It is indicated in di_nblocks.

The block allocation map control page, disk allocation map descriptor (dmap) control pages and dmap pages are all needed to rebuild at logredo time.

Overall, the following actions are taken at logredo time:

- Apply log record data to the specified page.
- Initialize freelist for directory B+ tree manager page or root.
- Rebuild inode allocation map manager.
- Rebuild block allocation map inode.

In addition, in order to ensure the log record applies only to a certain portion of page one time, logredo will start NoRedoFile,

The three log record types: REDOPAGE, NOREDOPAGE, NOREDOINOEXT, and UPDATEMAP, are the main force to initiate these actions.

If the aggregate has state of FM_DIRTY, then fsck.jfs will run after the logredo process since logredo could not get 100% recovery.

The maps are rebuilt in the following way: At the init phase, storage is allocated for the whole map file for both inode allocation map manager and block allocation map inode and then the map files are read in from the disk. The working allocation map (wmap) is initialized to zero. At the logredo time, the wmap is used to track the bits in persistent allocation map (pmap). In the beginning of the logredo process the allocation status of every block is in doubt. As log records are processed, the allocation state is determined and the bit of pmap is updated. This fact is recorded in the corresponding bits in wmap. So a pmap bit is only updated once at logredo time and only updated by the latest in time log record.

At the end of logredo, the control information, the freelist, etc. are built from the value of pmap; then pmap is copied to wmap and the whole map is written back to disk.

The status field s_state in the superblock of each file-system is set to FM_CLEAN provided the initial status was either FM_CLEAN or FM_MOUNT and logredo processing was successful. If an error is detected in logredo the status is set to FM_LOGREDO. The status is not changed if its initial value was FM_DIRTY. fsck should be run to clean up the probable damage if the status after logredo is either FM_LOGREDO or FM_DIRTY.

Log record Format

The log record has the following format:

<LogRecordData><LogRecLRD>

At logredo time, the log is read backwards. So for every log record JFS reads, LogRecLRD, which tells the length of the LogRecordData.

Logredo handles Extended Attributes (EA)

There is 16-byte EA descriptor which is located in the section I of dinode. The EA can be inline or outline. If it is inlineEA then the data will occupy the section IV of the dinode.

The dxd_t.flag will indicate so. If it is outlineEA, dxd_t.flag will indicate so and the single extent is described by EA descriptor. The section IV of dinode has 128 bytes. The xtroot and inlineEA share it. If xtree gets the section IV, xtree will never give it away even if xtree is shrunk or split. If inlineEA gets it, there is a chance that later inlineEA is freed and so xtree still can get it.

For outlineEA, FS will synchronously write the data portion so there is no log record for the data, but there is still an INODE log record for EA descriptor changes and there is a UPDATEMAP log record for the allocated pxd. If an outlineEA is freed, there are also two log records for it. One is INODE with EA descriptor zeroed out; another is the UPDATEMAP log record for the freed pxd.

For inlineEA, the data has to be recorded in the log record. It is not in a separate log record. Just one additional segment is added into the INODE log record. So an INODE log record can have at most three segments. When the parent and child inodes are in the same page, there is one segment for parent base inode; one segment for child base inode; and maybe one for the child inlineEA data.

More detail flow of Logredo

Below are the major steps that logredo must complete.

- Validate that the log is not currently in use.
- Recover if the JFS partition has increased in size.
- Open the log.
- Read the superblock and check the following fields: version, magic, state. If state is LOGREDONE, update the superblock and exit.
- Find the end of the log and initialize the data structures used by Logredo.
- Replay log. This reads the log backwards and processes records as it goes. Reading stops at the place specified by the first SYNCPT that is encountered.
- Start processing log records. There are only seven possible log records.
- After each loop through the different log records, check to see if the transaction just completed was the last for the current transaction, then flush the buffers.
- After processing all of the log records, check to see any 'dtpage extend' records were processed. If so go back and rebuild their freelists.
- Run logform so the following disk pages starting from the beginning of the log are formatted as follows:

  page 1 - log superlock
  page 2 - A SYNC log record is written
  page 3 - N - set to empty log pages.

- Flush data page buffer cache.
- Finalize the file system by updating the allocation map and superblock.
- Finalize the log by updating the following fields: end, state, and magic.

Next, all of the possible log records that the logredo utility must handle are described.

LOG_COMMIT record is used to insert the transaction ID (tid) from commit record into the commit array.

LOG_MOUNT record is the last record to be processed.

LOG_SYNCPT record is the log synch point.

LOG_REDOPAGE record contains information used to do the following operations:

- Update inode map for inodes allocated/freed.
- Update block map for an inode extent.
- Establish NoRedoFile or NoRedoExtent filters.
- Update block map for extents described in extent allocation descriptor B+ tree manager (xtree) root or node extent allocation descriptor list.

LOG_NOREDOPAGE record starts a NoRedoPage filter for xtree or dtree node.

LOG_NOREDOINOEXT record starts a NoRedoPage filter for each page in the inode extent being released.

LOG_UPDATEMP record is the update map log record, which describes the file system block extent(s) for the block map that needs to be marked.

Summary

When recovering after a system failure, JFS reads through the log file and if needed redoes each committed transaction. After redoing the committed transactions during a file system recovery, JFS locates all the transactions in the log file that were not committed at failure time and rolls back (undoes) each sub-operation that had been logged.

By logging JFS can restore the file system to a consistent state in a matter of seconds, by replaying the log file. By logging only meta-data changes JFS is able to keep the performance of this file system high.

Acknowledgements

The JFS Linux development team comprises:
Steve Best – sbest@us.ibm.com
Dave Kleikamp – shaggy@us.ibm.com
Barry Arndt – barndt@us.ibm.com

References

[JFSOverview]: “JFS overview” Steve Best, http://www-4.ibm.com/software/developer/ /library/jfs.html

[JFSLayout]: “JFS layout” Steve Best, Dave Kleikamp, http://www-4.ibm.com/software/developer/ /library/jfslayout/index.html

"JFS" Steve Best, published by Journal of Linux Technology April 2000 issue http://linux.com/jolt/

“Journal File Systems” Juan I. Santos Florido, published by Linux Gazette http://www.linuxgazette.com/issue55/florido.htm l

“Journaling Filesystems” Moshe Bar, published by Linux Magazine August 2000 issue http://www.linux-mag.com/2000-08/toc.html

“Journaling File Systems For Linux” Moshe Bar, published by BYTE.com May 2000 http://www.byte.com/column/servinglinux/BYT 20000524S0001

Source code for JFS Linux is available from http://oss.software.ibm.com/developerworks/ope nsource/jfs

JFS mailing list. To subscribe, send e-mail to majordomo@oss.software.ibm.com with “subscribe” in the Subject: line and “subscribe jfs-discussion” in the body.

Trademark and Copyright Information

Scalability and Failure Recovery in a Linux Cluster File System

Kenneth W. Preslan, Andrew Barry, Jonathan Brassow, Michael Declerck, A.J. Lewis, Adam Manthei, Ben Marzinski, Erling Nygaard, Seth Van Oort, David Teigland, Mike Tilstra, Steven Whitehouse, and Matthew O'Keefe

Sistina Software, Inc.
1313 5th St. S.E.
Minneapolis, Minnesota 55414
+1-612-379-3951, okeefe@sistina.com

Abstract

In this paper we describe how we implemented journaling and recovery in the Global File System (GFS), a shared-disk, cluster file system for Linux. We also present our latest performance results for a 16-way Linux cluster.

1 Introduction

Traditional local file systems support a persistent name space by creating a mapping between blocks found on disk drives and a set of files, file names, and directories. These file systems view devices as local: devices are not shared so there is no need in the file system to enforce device sharing semantics. Instead, the focus is on aggressively caching and aggregating file system operations to improve performance by reducing the number of actual disk accesses required for each file system operation [1], [2].

New networking technologies allow multiple machines to share the same storage devices. File systems that allow these machines to simultaneously mount and access files on these shared devices are called *shared file systems* [3], [4], [5], [6], [7]. Shared file systems provide a server-less alternative to traditional distributed file systems where the server is the focus of all data sharing. As shown in Figure 1, machines attach directly to devices across a *storage area network* [8], [9], [10].

A shared file system approach based upon a shared network between storage devices and machines offers several advantages:

1. *Availability* is increased because if a single client fails, another client may continue to process its workload because it can access the failed client's files on the shared disk.
2. *Load balancing* a mixed workload among multiple clients sharing disks is simplified by the client's ability to quickly access any portion of the dataset on any of the disks.
3. *Pooling* storage devices into a unified disk volume equally accessible to all machines in the system is possible, which simplifies storage management.
4. *Scalability* in capacity, connectivity, and bandwidth can be achieved without the limitations inherent in network file systems like NFS designed with a centralized server.

In the following sections we describe GFS version 4 (which we will refer to simply as GFS in the remainder of this paper), the current implementation including the details of our journaling code, new scalability results, changes to the lock specification, and our plans for GFS version 5, including file system versioning with copy-on-write semantics to support on-line backups.

2 GFS Background

Previous versions of GFS are described in the following papers: [7], [6], [11]. In this section we provide a sum-

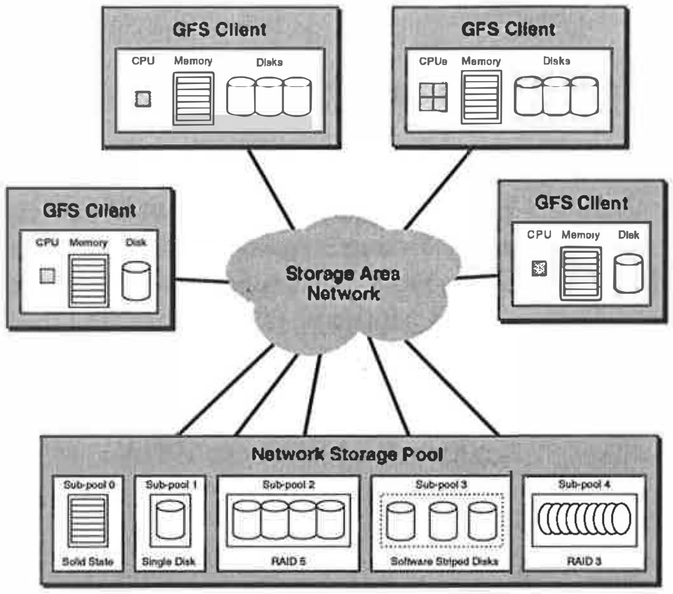


Figure 1: A Storage Area Network

mary of the key features of the Global File System.

2.1 DMEP

Device Memory Export Protocol (DMEP) is a mechanism used by GFS to synchronize client access to shared metadata. They help maintain metadata coherence when metadata is accessed by several clients. The DMEP SCSI command allows device to export memory to clients, and clients map lock state into these memory buffers. The lock state is contained in the storage devices (disks) and accessed with the SCSI DMEP command [12]. The DMEP command is independent of all other SCSI commands, so devices supporting the locks have no awareness of the nature of the resource that is locked (or even that the buffers are used to implement locks). The file system provides a mapping between file metadata and DMEP buffers.

Originally GFS used Dlocks SCSI command, which had the device maintaining the lock semantics as well as the lock state. The advantage of DMEP over Dlocks [13] is that the SCSI command is simpler, and the lock semantics may be modified on the client without affecting the SCSI command definition. This change was suggested by several SCSI disk vendors to ease implementation and create a more general SCSI synchronization primitive. All the semantics that used to be implemented by the SCSI devices as part of the Dlock command are now implemented on the client. The DMEP devices are just a reliable shared memory location.

In the event that a lock device is turned off and comes back on, all the state of the DMEP buffers on the device could be lost. Though it would be helpful if the locks were stored in some form of persistent storage, it is unreasonable to require it. Therefore, lock devices should not accept DMEP commands when they are first powered up. The devices should return failure results to all DMEP actions until a DMEP enable command is issued to the drive.

In this way, clients of the lock device are made aware that the locks on the lock device have been cleared, and can take action to deal with the situation. This is extremely important, because if machines assume they still hold locks on failed devices or on DMEP servers that have failed, then two machines may assume they both have exclusive access to a given lock. This inevitably leads to file system corruption.

The DMEP specification has been implemented as a server daemon called *memexpd* that can run on any UNIX machine, so that users need not have disk drives with special DMEP firmware to run GFS[14].

2.2 Lock Semantics

2.2.1 Expiration

In a shared disk environment, a failed client cannot be allowed to indefinitely hold whatever locks it held when it failed. Therefore, each client must periodically increment a counter in a DMEP buffer on the disk. If this counter (which the clients must monitor) isn't incremented for a given period of time, recovery functions can be started to free the lock state associated with that client. When a client fails to update its counter it is referred to as *timed-out*, and the act of updating the counter is often referred to as heartbeating the DMEP device.

2.2.2 Conversion Locks

The conversion lock is a simple, single-stage queue used to prevent writer starvation. In previous lock protocols used by GFS, one client may try to acquire an exclusive lock but fail because other clients are constantly acquiring and dropping the shared lock. If there is never a gap where no client is holding the shared lock, the writer requesting exclusive access never gets the lock. To correct this, when a client unsuccessfully tries to acquire a lock, and no other client already possesses that lock's conversion, the conversion is granted to the unsuccessful client. Once the conversion is acquired, no other clients can acquire the lock. All the current holders eventually unlock, and the conversion holder acquires the lock. All of a client's conversions are lost if the client expires.

2.3 Pool - A Linux Volume Driver

The Pool logical volume driver coalesces a heterogeneous collection of shared storage into a single logical volume. It was developed with GFS to provide simple logical device capabilities and to deliver DMEP commands to specific devices at the SCSI driver layer [15]. If GFS is used as a local file system where no locking is needed, then Pool is not required.

Pool also groups constituent devices into sub-pools. Sub-pools are an internal construction which does not affect the high level view of a pool[1] as a single storage device. This allows intelligent placement of data by the file system according to sub-pool characteristics. If one sub-pool contains very low latency devices, the file system could potentially place commonly referenced metadata there for better overall performance. There is not yet a GFS interface designed to allow this. Sub-pools are currently used in a GFS file system balancer [16]. The balancer moves files among sub-pools to spread data more evenly. Sub-pools now have an additional "type" designation to support GFS journaling. The file system requires that some sub-pools be reserved for journal space. Ordinary sub-pools will be specified as data space.

[1]The logical devices presented to the system by the Pool volume driver are called "pools".

2.4 File System Metadata

GFS distributes its metadata throughout the network storage pool rather than concentrating it all into a single superblock. Multiple resource groups are used to partition metadata, including data, dinode bitmaps and data blocks, into separate groups to increase file system scalability, avoid bottlenecks, and reduce the average size of typical metadata search operations. One or more resource groups may exist on a single device or a single resource group may include multiple devices.

Resource groups are similar to the Block Groups found in Linux's Ext2 file system. Like resource groups, block groups exploit parallelism and scalability by allowing multiple threads of a single computer to allocate and free data blocks; GFS resource groups allow multiple clients to do the same.

GFS also has a single block, the superblock, which contains summary metadata not distributed across resource groups, including miscellaneous accounting information such as the block size, the journal segment size, the dinode numbers of the two hidden dinodes and the root dinode, some lock protocol information, and versioning information.

Formerly, the superblock contained the number of clients mounted on the file system, bitmaps to calculate the unique identifiers for each client, the device on which the file system is mounted, and the file system block size. The superblock also once contained a static index of the resource groups which describes the location of each resource group and other configuration information. All this information has been moved to hidden dinodes (files).

There are two hidden dinodes:

1. *The resource index* – The list of locations, sizes, and glocks associated with each resource group

2. *The journal index* – The locations, sizes and glocks of the journals

This data is stored in files because it needs to be able to grow as the file system grows. In previous versions of GFS, we just allocated a static amount of space at the beginning of the file system for the Resource Index metadata, but this will cause problems when we expand the file system later. If this information is placed in a file, it is much easier to grow the file system at a later time, as the hidden metadata file can grow as well.

The Global File System uses Extendible Hashing [17], [7], [18] for its directory structure. Extendible Hashing (ExHash) provides a way of storing a directory's data so that any particular entry can be found very quickly. Large directories do not result in slow lookup performance.

2.5 Stuffed Dinodes

A GFS dinode takes up an entire file system block because sharing a single block to hold metadata used by multiple clients causes significant contention. To counter the resulting internal fragmentation we have implemented dinode stuffing which allows both file system information and real data to be included in the dinode file system block. If the file size is larger than this data section the dinode stores an array of pointers to data blocks or indirect data blocks. Otherwise the portion of a file system block remaining after dinode file system information is stored is used to hold file system data. Clients access stuffed files with only one block request, a feature particularly useful for directory lookups since each directory in the pathname requires one directory file read.

GFS assigns dinode numbers based on the disk address of each dinode. Directories contain file names and accompanying inode numbers. Once the GFS lookup operation matches a file name, GFS locates the dinode using the associated inode number. By assigning disk addresses to inode numbers GFS dynamically allocates dinodes from the pool of free blocks.

2.6 Flat File Structure

GFS uses a flat pointer tree structure as shown in Figure 2. Each pointer in the dinode points to the same height of metadata tree. (All the pointers are direct pointers, or they are all indirect, or they are all double indirect, and so on.) The height of the tree grows as large as necessary to hold the file.

The more conventional UFS file system's dinode has a fixed number of direct pointers, one indirect pointer, one double indirect pointer, and one triple indirect pointer. This means that there is a limit on how big a UFS file can grow. However, the UFS dinode pointer tree requires fewer indirections for small files. Other alternatives include extent-based allocation such as SGI's EFS file system or the B-tree approach of SGI's XFS file system [19]. The current structure of the GFS metadata is an implementation choice and these alternatives are worth exploration in future versions of GFS.

3 Improvements in GFS Version 4

We describe some of the recent improvements to GFS in the following sections.

3.1 Abstract Kernel Interfaces

We have abstracted the kernel interfaces above GFS, to the file-system-independent layer, and below GFS, to the block device drivers, to enhance GFS's portability. We hope to complete a port to FreeBSD sometime in 2001.

3.2 Fibre Channel in Linux

Until the summer of 1999, Fibre Channel (FC) support in Linux was limited to a single machine connected to a few drives on a loop. However, progress has been made in the quality of Fibre Channel fabric drivers and chipsets available on Linux. In particular, QLogic's QLA2100 and QLA2200 chips are supported in Linux, with multiple GPL'ed drivers written by QLogic and independent open source software developers. In testing in our laboratory with large Fabrics (32 ports) and large numbers of drives and GFS clients, the Fibre Channel hardware and software worked fine in static environments. Other FC card vendors like Interphase, JNI, Compaq, and Emulex are beginning to support Linux drivers for their cards.

However, it is possible to use GFS to share network disks exported through standard, IP-based network interfaces like Ethernet using Linux's Network Block Device software. In addition, new, fast, low-latency interfaces like

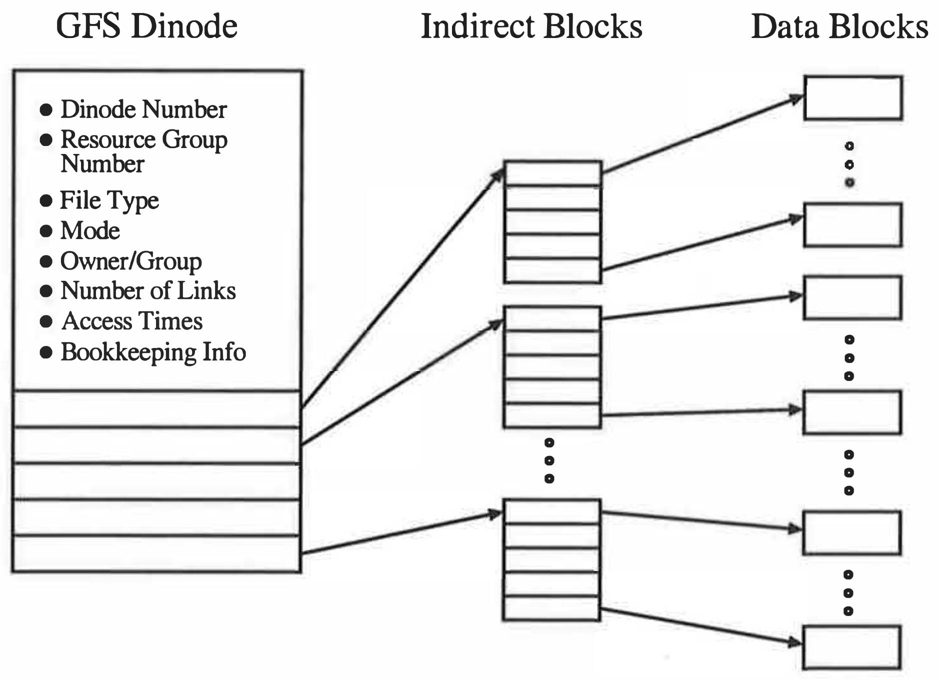


Figure 2: A GFS dinode. All pointers in the dinode have the same height in the metadata tree.

Myrinet combined with protocol layers like VIA hold the promise of high performance, media-independent storage networks.

3.3 Booting Linux from GFS and Context-Sensitive Symbolic Links

It has become possible to boot Linux from a GFS file system. In addition, GFS now supports context-sensitive symbolic links, so that Linux machines sharing a cluster disk can see the same file system image for most directories, but where convenient (such as /etc/???) can symbolically link to a machine-specific configuration file.

These two features help support a single system image by providing for a shared disk from which the machines in a cluster can boot up Linux, yet through context-sensitive symbolic links each machine can still maintain locally-defined configuration files. This simplifies system administration, especially in large clusters, where maintaining a consistent kernel image across hundreds of machines is a difficult task.

3.4 Global Synchronization in GFS

The lock semantics used in previous versions of GFS were tied directly to the SCSI Dlock command. This tight coupling was unnecessary, as the lock usage in GFS could be abstracted so that GFS machines could exploit any global lock space available to all machines. GFS-4 supports an abstract lock module that can exploit almost any globally accessible lock space, not just DMEP. This is important because it allows GFS cluster architects to buy any disks they like, not just disks that contain DMEP firmware.

A GFS lock module can implement callbacks to allow metadata caching and to improve lock acquisition latencies as shown in Figure 3. When client 2 needs a lock exclusively that is already held by client 1, client 2 first sends it's normal DMEP SCSI request to the disk drive (step 1 in the figure). This request fails and returns the list of holders, which happens to be client 1 (step 2). Client 2 sends a IP callback to client 1, asking 1 to give up the lock (step 3). Client 1 syncs all dirty (modified) data and metadata buffers associated with that lock to disk (step 4), and releases the lock. Client 2 may then acquire the lock (step 5).

Because clients can communicate with each other, they may hold locks indefinitely if no other clients choose to

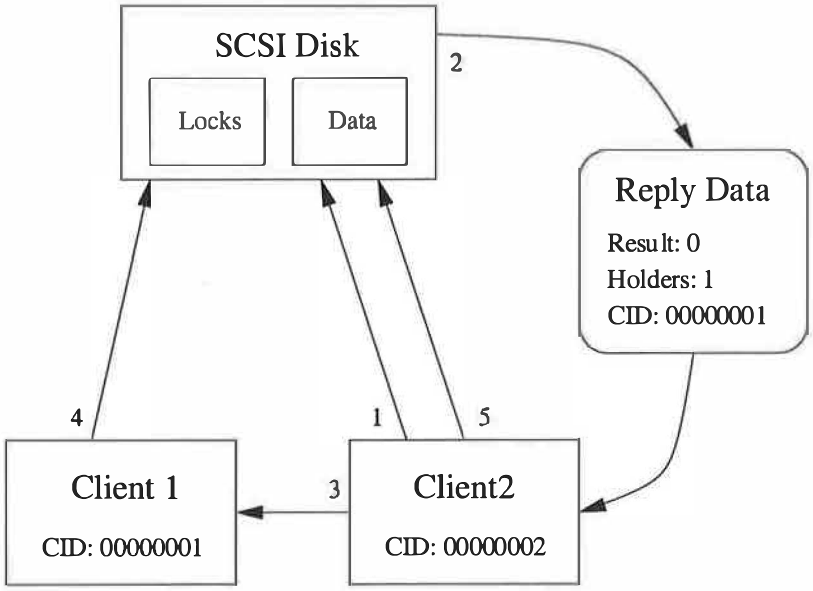


Figure 3: Callbacks on glocks in GFS

read from inodes associated with dlocks that are held. As long as a client holds a lock, it may cache any writes associated with the lock. Caching allows GFS to approach the performance of a local disk file system; our goal is to keep GFS within 10-15% of the performance of the best local, journaled Linux file systems across all workloads, including small file workloads.

In GFS-4, write caching is write-back, not write-through. GFS uses Global Locks (glocks). GFS uses interchangeable locking modules, some of which map glocks to DMEP buffers. Other locking methods, such as a distributed lock manager [9] or a centralized lock server, can also be used. GFS sees the Glocks as being in one of three states:

1. *Not Held* – This machine doesn't hold the Glock. It may or may not be held by another machine.
2. *Held* – this machine holds the Glock, but there is no current process using the lock. Data in the machine's buffers can be newer than the data on disk and there can be incore transactions for that glock. If another machine asks for the lock, the current holder will sync all the dirty transactions and buffers to disk and release the lock.
3. *Held + Locked* – the machine holds the Glock and there is a process currently using the lock. There can be newer data in the buffers than on disk. If another machine asks for the lock, the request is ignored temporarily, and is acted upon later. The lock is not released until the process drops the Glock down to the Held state.

When a GFS file system writes data, the file system moves the Glock into the Held+Locked state, acquiring the lock exclusively, if it was not already held. If another process is writing to that lock, and the Glock is already Held+Locked, the second process must wait until the Glock is dropped back down to Held.

The Write is then done asynchronously. The I/O isn't necessarily written to disk, but the cache buffer is marked dirty. The Glock is moved back to the Held state. This is the end of the write sequence.

The Buffers remain dirty until either bdflush or a sync causes the transactions and buffers to be synced to disk, or until another machine asks for the lock, at which point the data is synced to disk and the Glock is dropped to Not Held and the lock is released. This is important because it allows a GFS client to hold a Glock until another machine asks for it, and service multiple requests for the same Glock without making a separate lock request for each process.

3.5 GFS and Fibre Channel Documentation in Linux

We have developed documentation for GFS over the last year. Linux HOWTOs on GFS and Fibre Channel can be found at the GFS web page: http://www.globalfilesystem.org. In addition, there are conventional man pages for all the GFS and Pool Volume Manager utility routines, including mkfs, ptool, passemble, and pinfo[14].

4 File System Journaling and Recovery in GFS

To improve performance, most local file systems cache file system data and metadata so that it is unnecessary to constantly touch the disk as file system operations are performed. This optimization is critical to achieving good performance as the latency of disk accesses is 5 to 6 orders of magnitude greater than memory latencies. However, by not synchronously updating the metadata each time a file system operation modifies that metadata, there is a risk that the file system may be inconsistent if the machine crashes.

For example, when removing a file from a directory, the file name is first removed from the directory, then the file dinode and related indirect and data blocks are removed. If the machine crashes just after the file name is removed from the directory, then the file dinode and other file system blocks associated with that file can no longer be used by other files. These disk blocks are now erroneously now marked as in use. This is what is meant by an inconsistency in the file system.

When a single machine crashes, a traditional means of recovery has been to run a file system check routine (fsck) that checks for and repairs these kinds of inconsistencies. The problem with file system check routines is that (a) they are slow because they take time proportional to the size of the file system, (b) the file system must be off-line while the fsck is being performed and, therefore, this technique is unacceptable for shared file systems. Instead, GFS uses a technique known as file system journaling to avoid fsck's altogether and reduce recovery time and increase availability.

4.1 The Transaction Manager

Journaling uses transactions for operations that change the metadata state. These operations must be atomic, so that the file system moves from one consistent on-disk state to another consistent on-disk state. These transactions generally correspond to VFS operations such as create, mkdir, write, unlink, etc. With transactions, the file system metadata can always be quickly returned to a consistent state.

A GFS journaling transaction is composed of the metadata blocks changed during an atomic operation. Each journal entry has one or more locks associated with it, corresponding to the metadata protected by the particular lock. For example, a creat() transaction would contain locks for the directory, the new dinode, and the allocation bitmaps. Some parts of a transaction may not directly correspond to on-disk metadata.

All metadata blocks contain a generation number that is incremented each time it is changed, and that is used in recovery.

A transaction is created in the following sequence of steps:

1. start transaction
2. acquire the necessary Glocks
3. check conditions required for the transaction
4. pin the in-core metadata buffers associated with the transaction (i.e., don't allow them to be written to disk)
5. modify the metadata
6. pass the Glocks to the transaction
7. commit the transaction by passing it to the Log Manager

To represent the transaction to be committed to the log, the Log Manager is passed a structure which contains a list of metadata buffers. Each buffer knows its Glock number. Passing this structure represents a commit to the in-core log.

4.2 The Log Manager

The Log Manager is separate from the transaction module. It takes metadata to be written from the transaction module and writes it to disk. The Transaction Manager pins, while the Log Manager unpins. The Log Manager also manages the Active Items List, and detects and deals with Log wrap-around.

For a shared disk file system, having multiple clients share a single journal would be too complex and inefficient. Instead, as in Frangipani [4], each GFS client gets its own journal space, that is protected by one lock that is acquired at mount time and released at unmount (or crash) time. Each journal can be on its own disk for greater parallelism. Each journal must be visible to all clients for recovery.

In-core log entries are committed asynchronously to the on-disk log. The Log Manager follows these steps:

1. get the transaction from the Transaction Manager
2. wait and combine this transaction with others (asynchronous logging)
3. perform the on-disk commit
4. put all metadata in the Active Items List
5. unpin the metadata
6. later, when the metadata is on disk, remove it from the Active Items List

Recall that all transactions are linked to one or more Glocks, and that Glocks may be requested by other machines during a callback operation. Hence, callbacks may result in particular transactions being pushed out of the in-core log and written to the on-disk log. Before a Glock is released to another machine, the following steps must be taken:

1. transactions dependent on that Glock must be flushed to the log
2. the in-place metadata buffers must be synced
3. the in-place data buffers must be synced

Only transactions directly or indirectly dependent on the the requested Glock need to be flushed. A journal entry is dependent on a Glock if either (a) it references that Glock directly, or (b) it has Glocks in common with transactions which reference that Glock directly.

For example, in Figure 4, five transactions in sequential order (starting with 1) are shown, along with the Glocks upon which each transaction is dependent. If Glock 6 is requested by another machine, transactions 1, 2, and 5 must be flushed to the on-disk log in order. (Because transactions involving overlapping glocks are combined as they are commited to the in-core log, transaction 3 will be written out as well. It's not strictly necessary, though.) Then the in-place metadata and data buffers must be synced for Glock 6, and finally Glock 6 is released.

4.3 Recovery

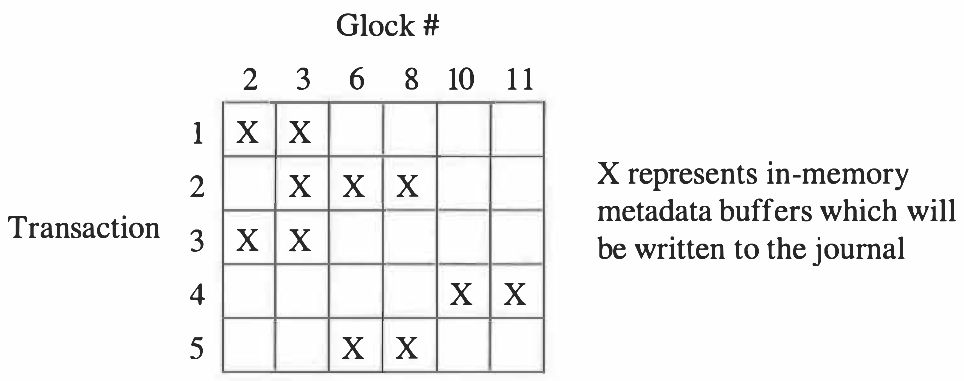
Journal recovery is initiated by clients in two cases:

- a mount time check shows that any of the clients were shutdown uncleanly or otherwise failed
- a locking module reports an expired client when it polls for expired machines

In each case, a recovery kernel thread is called with the expired client's ID. The machine then attempts to begin recovery by acquiring the journal lock of a failed client. A very dangerous special case can result when a client (known as a zombie) fails to heartbeat its locks, so the other machines think it is dead, but it is still alive; this could happen, for example, if for some reason the "failed" client temporarily was disconnected from the network. This is dangerous because the supposedly failed client's journal will be recovered by another client, which has a different view of the file system state. This "split-brain" problem will result in file system corruption. For this reason, the first step in recovery after acquiring the journal lock of a failed client is to prevent the failed machine from writing to the shared device. This operation is called I/O Fencing.

There are several methods for I/O Fencing.

- *Network Power Switch* – a machine connects to a power switch over IP and asks it to cycle the power on a failed client
- *X10* – a machine can piggy-back signals onto of the in-the-wall-power to cause another machine's power to be cycled.
- *Persistent Reservation* – Some SCSI devices implement a command that allows one machine ask the device to ignore another machine.
- *Fibre Channel Zoning* – Some Fibre Channel switches support the capability to prevent the failed client from accessing the disks.



Glock #

Transaction	2	3	6	8	10	11
1	X	X				
2		X	X	X		
3	X	X				
4					X	X
5			X	X		

X represents in-memory metadata buffers which will be written to the journal

Figure 4: Journal Write Ordering Imposed by Lock Dependencies During GFS Lock Callbacks

GFS allows arbitrary I/O fencing methods to disable the failed clients access to the shared storage devices. There are currently modules that support all of the above methods except Persistent Reservation. (Persistent Reservation is only just starting to appear on SCSI devices.)

Once a client fences out the fail machine and obtains its journal lock, journal recovery proceeds as follows: the tail (start) and head (end) entries of the journal are found. Partially-committed entries are ignored. For each journal entry, the recovery client tries to acquire all locks associated with that entry, and then determines whether to replay it, and does so if needed. All expired locks are marked as not expired for the failed client. At this point, the journal is marked as recovered.

The decision to replay an entry is based on the generation number in the metadata found in the entry. When these pieces of metadata are written to the log, their generation number is incremented. The journal entry is replayed if the generation numbers in the journal entry are larger or equal to the in-place metadata.

Note that machines in the GFS cluster can continue to work during recovery unless they need a lock held by a failed client.

4.4 Comparison to Alternative Journaling Implementations

The main difference between journaling in a local file system and GFS is that GFS must be able to flush out transactions in an order other than that in which they were created. A GFS client must be able to respond to callbacks on locks from other clients in the cluster. The client should then flush only the transactions that are dependent on that lock. This means that GFS doesn't automatically combine transactions as they are commited in-core. They are only combined if they share glocks.

4.5 Cluster Configuration

There are a number of identifiers that each member of the cluster needs to know about all the other members: Hostname, IP address, Journal Number, and the I/O fencing method. The mappings for each host in the cluster are stored outside the file system by the lock module. For the DMEP lock module, these values are stored on a block device readable by all machines in the cluster.

4.6 Online Growing of File Systems

It is important to able to add more storage space to a running cluster. When A new feature of GFS 4 is that it now supports the ability to grow the file system on-line. One feature of FC is that disks can be added to a Storage Area Network (SAN). Once that is done the disks can be added to the file system's Pool. The process of growing the Pool happens in three steps:

1. Labels describing how the new disks fit into the Pool are written to the new disks.
2. An atomic write to the label on the first disk in the Pool sets the Pool to its new size.
3. At this point, the Pool can be reassembled and the new space can be used.

One interesting feature of Pool is that step 3 doesn't need to happen on all the machines right away. The reassembly to access the new storage space only happens when that new space is needed. As the Pool driver in each machine maps block request to individual disks, it looks for block numbers that doesn't exist in the current Pool. When it sees that the block number being accessed is too large, the Pool labels are reread from the disks and the new, bigger Pool is reassembled. Then GFS can be informed that new space is available and it can be added to the file system. The new space takes the form of new Resource Groups at the end of the file system. The locations of these new resource groups are written to the end of the Resource Index hidden dinode.

Again, the growth of the GFS file system is automatically detected by other machines in the cluster. During normal operation all the machines hold a shared lock on the Resource Index dinode. The process growing the file system acquires that lock exclusively when adding the new resource groups to the dinode. After that, each machine can reacquire its shared lock on the dinode and reread the new index. Each machine can then access the new storage space.

5 Scalability

Figure 5 shows one to sixteen GFS hosts being added to a constant size file system and each performing a workload of a million random operations. (These results are fore a previous non-journaled version of GFS.) These sixteen machines were connected across a Brocade Fabric fabric to 8 8-disk enclosures, each configured as a single 8-disk loop. The workload consisted of 50 percent reads, 25 percent appends/creates and 25 percent unlinks. Each machine was working in in its own directory and the directories were optimally placed across the file system. Notice that the scalability curve shows nearly perfect speedup. These new results compare favorably with the dismal scaling results obtained for the early versions of GFS [6], which didn't cache locks, file data, or file system metadata.

6 Conclusions and Future Work

In this paper, we described the GFS journaling and recovery implementation and other improvements in GFS version 4 (GFS-4). These include a lock abstraction and network block driver layer, which allow GFS to work with almost any global lock space or storage networking media. The new DMEP specification simplifies the work required by SCSI storage vendors, and allows the lock semantics to be refined over time. In addition, a variety of other changes to the file system metadata and pool volume manager have increased both performance and flexibility. Taken together, these changes mean that GFS can now enter a beta test phase as a prelude to production use. Early adopters who are interested in clustered file systems for Linux are encouraged to install and test GFS to help us validate its performance and robustness.

With the work on journaling and recovery complete, we intend to look at several new features for GFS. These include file system versioning for on-line snapshots of file system state using copy-on-write semantics. File system snapshots allow a slightly older version of the file system to be backed up on-line while the cluster continues to operate. This is important in high-availability systems.

References

[1] L. McVoy and S. Kleiman. Extent-like performance from a unix file system. In *Proceedings of the 1991 Winter USENIX Conference*, pages 33–43, Dallas, TX, June 1991.

[2] Uresh Vahalia. *Unix Internals: The New Frontiers*. Prentice-Hall, 1996.

[3] Roy G. Davis. *VAXCluster Principles*. Digital Press, 1993.

[4] Chandramohan Thekkath, Timothy Mann, and Edward Lee. Frangipani: A scalable distributed file system. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 224–237, October 1997.

[5] Matthew T. O'Keefe. Shared file systems and fibre channel. In *The Sixth Goddard Conference on Mass Storage Systems and Technologies in cooperation with the Fifteenth IEEE Symposium on Mass Storage Systems*, pages 1–16, College Park, Maryland, March 1998.

[6] Steve Soltis, Grant Erickson, Ken Preslan, Matthew O'Keefe, and Tom Ruwart. The design and performance of a shared disk file system for IRIX. In *The Sixth Goddard Conference on Mass Storage Systems and Technologies in cooperation with the Fifteenth IEEE Symposium on Mass Storage Systems*, pages 41–56, College Park, Maryland, March 1998.

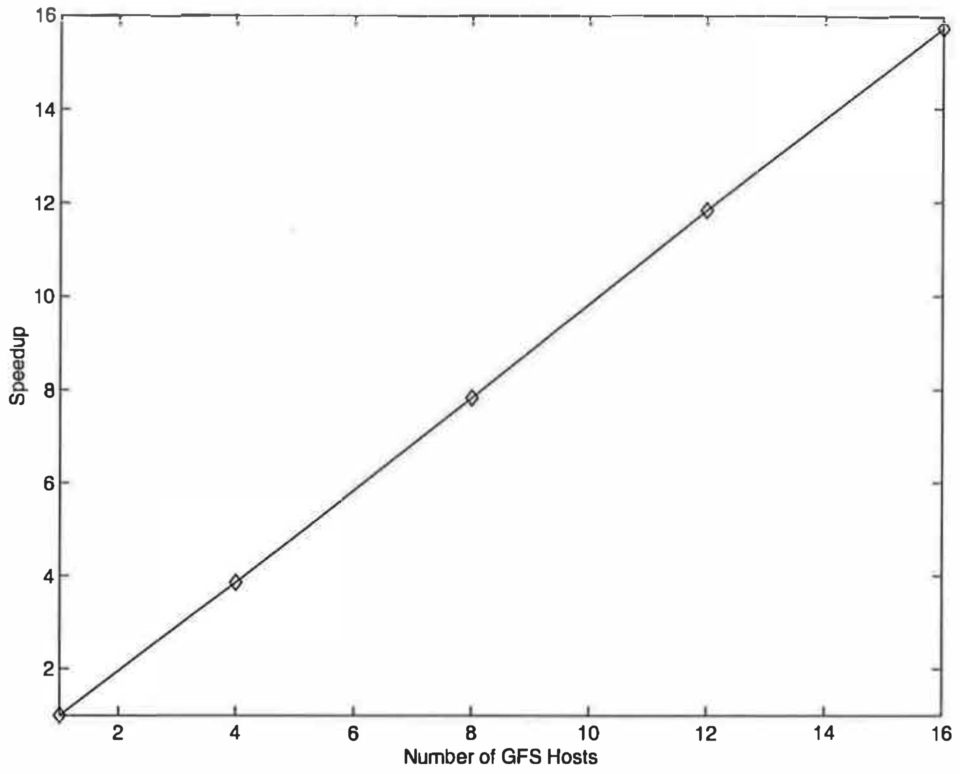


Figure 5: Sixteen machine speedup for independent operations

[7] Kenneth W. Preslan et al. A 64-bit, shared disk file system for linux. In *The Seventh NASA Goddard Conference on Mass Storage Systems and Technologies in cooperation with the Sixteenth IEEE Symposium on Mass Storage Systems*, pages 22–41, San Diego, CA, March 1999.

[8] Alan F. Benner. *Fibre Channel: Gigabit Communications and I/O for Computer Networks*. McGraw-Hill, 1996.

[9] N. Kronenberg, H. Levy, and W. Strecker. VAXClusters: A closely-coupled distributed system. *ACM Transactions on Computer Systems*, 4(3):130–146, May 1986.

[10] K. Matthews. Implementing a Shared File System on a HiPPi disk array. In *Fourteenth IEEE Symposium on Mass Storage Systems*, pages 77–88, September 1995.

[11] Steven R. Soltis, Thomas M. Ruwart, and Matthew T. O'Keefe. The Global File System. In *The Fifth NASA Goddard Conference on Mass Storage Systems and Technologies*, volume 2, pages 319–342, College Park, Maryland, March 1996.

[12] The GFS Group. Device memory export protocol specification. http:// www.globalfilesystem.org/ Pages/ dmep.html, August 2000.

[13] Ken Preslan et al. Dlock 0.9.6 specification. http:// www.globalfilesystem.org/ Pages/ dlock.html, May 2000.

[14] Mike Tilstra et al. The gfs howto. http://www.globalfilesystem.org /howtos/ gfs_howto.

[15] David Teigland. The pool driver: A volume driver for sans. Master's thesis, University of Minnesota, Department of Electrical and Computer Engineering, Minneapolis, Minnesota, October 1999. http://www.globalfilesystem.org/ Pages/ theses.html.

[16] Manish Agarwal. A Filesystem Balancer for GFS. Master's thesis, University of Minnesota, Department of Electrical and Computer Engineering, Minneapolis, Minnesota, June 1999. http://http://www.globalfilesystem.org /Pages/ theses.html.

[17] Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. Extendible Hashing –

a fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3):315–344, September 1979.

[18] Michael J. Folk, Bill Zoellick, and Greg Riccardi. *File Structures*. Addison-Wesley, March 1998.

[19] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the xfs file system. In *Proceedings of the USENIX Annual Technical Conference*, pages 1–14, San Diego, CA, January 1996.

All GFS publications and source code can be found at `http://www.globalfilesystem.org`.

VA SystemImager

Brian Elliott Finley <brian@valinux.com>

Abstract

Linux use in corporations and research organizations has been growing at an amazing rate. It is often used on large numbers of identical systems serving as Internet server farms or high performance computing clusters. Without the help of specialized tools, the time and effort required to install and maintain large numbers of machines grows almost linearly as new systems are added. This creates the demand for a tool with the ability to automate the installation of new systems and maintain the software, configuration, and content of those systems on an ongoing basis.

System administrators at large sites will often develop tools for automating the deployment and update of their own systems, but these tools are often very inflexible and are only designed to address the specific needs of one particular set of systems. Therefore these tools are often re–created by system administrators at site after site, not being able to capitalize on the work of their neighbors. The need was perceived for a tool that could provide this functionality at many different sites with different configurations. This required that the tool be easy to install, simple and straightforward to use, and that it be designed in an open and extensible manner to accommodate future changes and site specific customizations.

This paper describes the resultant tool, VA SystemImager. It is Open Source software and is designed in a very modular manner. Great pains were taken to ensure that it would be flexible and could easily be modified to accommodate new hardware, software, and site specific configuration needs in future iterations. VA SystemImager is written mostly in Perl and makes use of rsync(1), syslinux(2), and pxelinux(2). It also required the creation of a customized miniature Linux distribution for the installation media. This paper will also discuss some of the differences between VA SystemImager and the KickStart network installation tool from RedHat. KickStart is the tool most often compared to VA SystemImager. Although there are a handful of other tools available, none of them offer the flexibility and ease of use of VA SystemImager.

Extended Abstract

Design Goals

– Images should be pulled from an already running system.
– Completely unattended installs were a must.
– The unattended install system had to be able to repartition the destination drive(s).
– One of the main design goals was ease of use. This had to be a tool that could be used by a system administrator that didn't necessarily understand how it worked.
– It was also necessary for it to install easily and quickly so that it could be useful right away without a lot of site specific customization.
– Images should be stored as normal files in appropriately named directories as opposed to "dd" style block level images of physical disks.
– Not everyone who needs to install a lot of Linux systems uses the same distribution, so it had to be independent of any and all packaging systems (such as RPM).
– It should be able to store multiple images, for different types of systems and for revision control, and provide a mechanism for unattended install clients to know which image to install.
– Once a client was installed, it should be able to update itself to a new or updated image.
– It should easily accommodate different distributions.
– It should have a command line interface designed such that it could later be wrapped with a GUI.

Some VA SystemImager terms and commands that will be referred to in this abstract:

– autoinstall client – A machine on to which Linux is to be installed using the VA SystemImager automated process.
– autoinstall media – The media that is used to boot an autoinstall client in order to begin the autoinstall process. This media can be a floppy, a CDROM, the network, or the local hard drive of the autoinstall client.
– updateclient – A command that is executed on client systems allowing them to be updated or synchronized to a new or updated image.
– imageserver – The machine that will hold the images.
– master client – The machine that has been manually

installed and configured the way you want your image to look.
– getimage – This command is run from the imageserver to pull a system image from a master client.
– prepareclient – This command is run on the master client immediately prior to running getimage on the imageserver.
– makedhcpserver – Used to create the /etc/dhcpd.conf file. DHCP can be used to assign IP addresses to autoinstall clients.
– makedhcpstatic – Used to re–write the /etc/dhcpd.conf file, adding static entries based on the IP addresses already handed out to all of the autoinstall clients.
– autoinstall script – A unique autoinstall script is created for each image and is used by the autoinstall client as part of the autoinstall process. The names of these scripts begin with the image name and end in .master. For example: "my_webserver_image_v1.master"
– addclients – Creates a soft link to the master autoinstall script with the name of each host that will receive that image. It also allows you to populate the /etc/hosts file with sequential host names and IP addresses.

Resultant Architecture

VA SystemImager began as a series of utilities written in bash. Minimal system requirements were considered a top priority. As VA SystemImager matured, and the utilities became internally more complex, it became clear that bash was falling short of the need. Perl was chosen to pick up the yoke and has allowed for cleaner, more advanced code. It was determined that Perl is installed as part of most "base" Linux installs and therefore was a reasonable choice from a minimal requirements perspective.

The architecture was designed to be open to future modification at every level. The protocol used for transferring files during installs and updates is currently rsync(1). But the modular code will easily allow for a drop–in replacement using mftp(5) or other appropriate file transfer utilities. All file transfer mechanisms are implemented in a "pull" fashion, which is generally considered to be superior to a "push". Using a "pull" mechanism, it is much easier to monitor the state of the receiving system prior to and during the file transfers.

There are other methods available for doing automatic installs, such as RedHat's KickStart which installs systems based on a list of pre–defined packages. But package based installs are very limiting in that they generally don't have an automated way for dealing with non–packaged files. If you re–compile your kernel, add a piece of non–packaged software, or modify certain configuration files, you are usually required to do some sort of scripting or programming to deal with these "special cases".

In order keep imaging simple, VA SystemImager uses images that are based on a working installed system. We call this system a "master client". Just get one of your machines working exactly the way you want and pull it's image to the imageserver with the "getimage" command. You can re–compile your kernel, install custom software, and do any configuration file tweaking you like. VA SystemImager will get it all.

Now that you have your master client configured, we need to run the "prepareclient" command. prepareclient will collect the partition information from your disks and put it in the /etc/partitionschemes directory. A file will be created in this directory for each of your disks and will contain that disks partition information. prepareclient will also creates an rsync(1) configuration file (/etc/rsyncd.conf) and starts rsync in server mode (rsync ––daemon). This allows the imageserver to pull the image from the client, but will not cause the rsync daemon to be restarted after the master client is rebooted. This helps avoid security concerns of sharing a master client's root filesystem via rsync. rsync has the ability to use OpenSSH(6) as an alternate shell and plans are in place to modify VA SystemImager to run all operations over OpenSSH(6) for security purposes.

On the imageserver we now run the getimage command. Here's an example: "getimage –master–client=192.168.1.1 –image=my_webserver_image_v1" getimage contacts the master client and requests it's /etc/mtab file. This file contains the list of mounted filesystems and the devices on which they are mounted. It pulls out the mount points for the filesystems that are unsupported and creates an exclusion list. Currently supported filesystems are ext2 and reiserfs. Unsupported filesystems are things like proc, devpts, iso9660, etc. getimage then pulls the master client's entire system image, excluding the filesystems in the exclusion list. The files are pulled by connecting to the rsync(1) daemon running on the master client. All the files from the client will be copied over, recreating the filesystem and directory hierarchy in the image directory.

getimage can also be used to update an existing image. By simply specifying an existing image name, you are asking getimage to update that image to match the files on you master client. In this case, only the files that are different will be copied over. Files that exist in the old image but not on the master client will be deleted, and files that exist in both places but have changed will be updated. This is one way to keep an image updated when new security patches or other system updates come out. However, the recommended method is to never overwrite a known working image, so that you have a form a revision control. This is not true revision control, where individual file revisions are tracked on a line by line basis. It is more of a revision control on an image by image basis. This form of revision control also ties in to the updateclient command which will be discussed later. By default, all images are stored in the parent directory of "/var/spool/systemimager/images/" in a directory that bears the image name. For example: "/var/spool/systemimager/images/my_webserver_image_v1/".

After getimage has pulled the files to the image directory on the imageserver it creates a customized autoinstall script. The master script in this case would be named "my_webserver_image_v1.master". All autoinstall scripts are placed in the "/tftpboot/systemimager/" directory. The disk partitioning information left behind by the prepareclient command is used to add the necessary commands to re–partition the disk(s) on the autoinstall clients. Filesystem information taken from the /etc/fstab file in the image (Ie.: "/var/spool/systemimager/images/my_webserver_image_v1/etc/fstab") and is used to determine the appropriate filesystem creation commands and to determine mount points for the autoinstall process. Based on command line options passed to getimage or questions it has asked, certain networking information is added to the autoinstall script. This information is added in variable form as the autoinstall client will later determine the values for things such as it's hostname and IP address.

When running getimage interactively, it will prompt you to run the addclients command. addclients will ask you for the series of hostnames that you will be installing by combining a base host name and a number range. For example, if your base host name is "www", and your number range is from "1" to "3", then the resultant host names would be "www1, www2, www3". It will then prompt you to choose the image that will be installed to these hosts and will create soft links for each hostname that point to the master script for that image. For example: "www3.sh –> web_server_image_v1.master". If the image is updated and you choose to allow getimage to also update the master autoinstall script, then each of the associated soft links therefore point to the new master script. If individual host configuration is necessary, the soft link for that host can be removed and replaced with a copy of the master script that can then be customized for that host. This customization is a manual process and is up to the administrator of the system. addclients will then prompt you for the IP address information for these hosts and will re–write the imageserver's /etc/hosts file accordingly and copy this file to /tftpboot/systemimager/hosts. The latter file is used during the autoinstall process if the clients are using DHCP to obtain their IP addresses.

The unattended install portion is flexible and can work with most any hardware available. It is also easily modified to work with new or special hardware. A miniature Linux distribution is used for the boot media for "autoinstalls" (unattended installs). It consists of a customized kernel and an initial ram disk. The same kernel and initial ram disk (initrd.gz) can be used to boot off floppy disks, CDROMs, the network, or a running system's local hard drive. The commands "makeautoinstalldiskette" and "makeautoinstallcd" make use of the syslinux(2) utility to create floppies and CDROMs that will boot the VA SystemImager kernel and initial ram disk. pxelinux(2), which is a sister tool to syslinux(2), allows the same kernel and initial ram disk to boot PXE capable machines off the network. A configuration file is needed by syslinux(2) and by pxelinux(2), but VA SystemImager handles this for you and the two tools are able to use the same configuration file.

The autoinstall client is a miniature Linux distribution that has been customized to contain the specific commands and utilities necessary to perform autoinstalls to clients. The kernel is compiled to contain all the necessary drivers for a majority of systems. Custom kernels can be compiled to match special configurations. To use a custom compiled kernel, simply copy it to /tftpboot/kernel. All of the autoinstall media is created from /tftpboot/kernel and /tftpboot/initrd.gz. syslinux is used to load the initial ram disk and to boot the kernel when using an autoinstall diskette or an autoinstall CD. pxelinux is used to load the initial ram disk and to boot the kernel when using network booting.

Once the kernel has booted, it mounts the initial ram disk as it's root filesystem. It then executes an initialization script that has been customized to do VA SystemImager specific things. This script will use DHCP to get the autoinstall client's IP address information. It makes the assumption that the DHCP server is the imageserver and contacts it to request the utilities that would not fit in the initial ram disk. It copies these utilities to another ram disk that is mounted as /tmp1. It then requests a hosts file from the imageserver (the one in /tftpboot/systemimager) and parses this file to find it's IP address in order to determine it's hostname. Finally it requests an autoinstall script from the imageserver based on this hostname and executes it. The autoinstall script is image specific. This is how a client determines which image it will receive.

The most common way to assign IP addresses to the autoinstall clients is DHCP. To easify the configuration of the DHCP configuration file (/etc/dhcpd.conf) VA SystemImager includes a utility called makedhcpserver. makedhcpserver will prompt you for all the necessary information to create a DHCP configuration file that is appropriate for VA SystemImager. It is also possible to continue to use DHCP to assign static IP addresses to your clients after installation. If you choose to do so, simply run the makedhcpstatic command. It will rewrite your /etc/dhcpd.conf file on the imageserver to contain static entries for each of your hosts.

Alternately, hostname, imageserver, and networking information can be put in a configuration file on a floppy diskette. When the autoinstall client boots, it will look for this file on the floppy and use the provided values instead of determining them dynamically. This will work with any of the autoinstall media. The configuration file can even be put on the autoinstall floppy itself! The format of this configuration file is simply VARIABLE=value for all the appropriate variables. The name of this file must be local.cfg and it must exist on the root of the floppy. The floppy can be formatted with either ext2 or fat. An example local.cfg file can be found with the documentation files which are installed in /usr/doc.

Sometimes you will want to update an image on your imageserver. There are a couple of ways to do this. The first way is do directly edit the files in the image directory. The best way to do this is to chroot into the image directory. Once you have done the chroot, you can work with the image as if it were actually a running machine. You can even install packages with RPM, for example. The second way is to run the getimage command again, specifying a master client that has been modified in the desired way. Only the files that have changed will be pulled across. Files that have been deleted on the master client will also be deleted in the image. You are also given the option to update the master autoinstall script for the image or to leave it alone. The advantages of this method are that you can verify that your new configuration works on the master client, and that the master autoinstall script is updated.

Once a system has been autoinstalled, the updateclient command can be used to update a client system to match a new or updated image on the imageserver. Let's say that you've installed your companies 300 web servers and a security patch comes out the next day. You simply update the image on the imageserver and run updateclient on each of your 300 web servers. Only the modified files are pulled over, and your entire site is patched! It is recommended that you create an entirely new image with a new version number so that you have a form of revision control. This way, if you find out that the patch you applied hosed your entire web farm, you simply do an updateclient back to the know working image!

By incorporating some modifications sent in by A.L. Lambert, using the "updateclient" command with the –autoinstall option will copy the autoinstall kernel and initial ram disk to the local hard drive of the client. It will then re–write the /etc/lilo.conf file to include an appropriate entry for the new kernel and initial ram disk and specify this new kernel as the default using the "–D" option. The next time the client system is booted, it will load the VA SystemImager kernel and initial ram disk, which will begin the autoinstall process! This means that you can re–install any running Linux machine without having to have someone feed the machine a floppy or CD, and without having to reconfigure the BIOS to boot off the network (which can be quite squirrelly with some BIOSes).

Summary of Steps

1) Install the VA SystemImager software on the machine chosen to be the imageserver and configure the machine to be a DHCP server using the "makedhcpserver" command.
2) Choose the "master client" and make any desired changes to this system including recompiling the kernel, installing software, and tweaking configuration files.
3) Install the VA SystemImager client software on the client, which prepares it for having it's image pulled.
4) Issue the "getimage" command from the imageserver, specifying the master client and the name to assign to the resultant image.
5) Create autoinstall media for floppy installs using the "makeautoinstalldiskette" command, or for CD installs using the "makeautoinstallcd" command. Network boot clients should be told to boot off the network.
6) Boot the autoinstall client off the chosen media and watch it autoinstall!

VA SystemImager is available for download from:

http://systemimager.sourceforge.net/

Future Improvements

VA SystemImager is under active development. Many new features are being added by the core developers and by end users. Some of the more notable future improvements are:

– Support for software RAID on autoinstall clients.
– A multicast install utility is being developed that will allow an unlimited number of autoinstall clients to be installed in parallel.
– See http://systemimager.sourceforge.net/TODO for others.

Acknowledgements

VA SystemImager was conceived and developed by Brian Finley. It's initial implementation was known as pterodactyl and was used for software and password updates to Solaris boxes of varying hardware and OS versions across a nationwide enterprise network. Over time it evolved into the Linux specific autoinstall and update tool that it is today. Many of the design decisions for VA SystemImager were based on perceived shortcomings in other automated install tools for systems such as Solaris, RedHat Linux, and Windows.

Other people who have contributed code or documentation that has been incorporated (alphabetical order):
– Susan Coghlan <smc@acl.lanl.gov>
– Paonia Ezrine <paonia@home.welcomehome.org>
– Michael Jennings <mej@valinux.com>
– Ari Jort <ajort@valinux.com>
– Ian McLeod <ian@valinux.com>
– Michael P. McLeod <mmcleod@bcm.tmc.edu>
– Michael R. Nolta <mrnolta@princeton.edu>
– Laurence Sherzer <lsherzer@gate.net>
– Wesley Smith <wessmith@engr.sgi.com>

Other people who have contributed ideas and suggestions (alphabetical order):
– Ted Arden <ted@valinux.com>
– Tanmoy Bhattacharya <tanmoy@lanl.gov>
– Susan Coghlan <smc@acl.lanl.gov>
– Steven Duchene <sad@valinux.com>
– Stephen Greene <sgreene@valinux.com>
– Bartosz Ilkowski <barbi@danforthcenter.org>
– Ari Jort <ajort@valinux.com>
– Brian Luethke <luethke@msr.epm.ornl.gov>
– Chip Salzenberg <chip@valinux.com>
– Robert Saft <zardoz@valinux.com>

References

(1) rsync –– remote synchronization/update protocol. rsync is used to transfer files from the imageserver to an autoinstall client or a client using the "updateclient" command.

rsync was written by Andrew Tridgell <tridge@samba.org> and Paul Mackerras <Paul.Mackerras@cs.anu.edu.au>.

http://rsync.samba.org/

(2) syslinux –– syslinux is a boot loader for the Linux operating system. It is used with the autoinstall diskette.

syslinux was written by H. Peter Anvin.

ftp://ftp.us.kernel.org/pub/linux/utils/boot/syslinux/

(3) pxelinux –– pxelinux is a network boot loader for the Linux operating system. It is used for autoinstalling over the network.

pxelinux was written by H. Peter Anvin and is part of the syslinux package listed above.

(4) tftp–hpa –– tftp–hpa is a tftp server that has been modified to accept certain options, or "non–options" required for the broken PXE protocol.

tftp–hpa was written by H. Peter Anvin.

http://www.kernel.org/pub/software/network/tftp/

(5) mftp –– mftp is a multicast ftp client that is currently being written by Ian McLeod <ian@valinux.com>. It is based on the multicast libraries being written by Roland Dreier <roland@valinux.com>.

(6) OpenSSH –– Ssh (Secure Shell) a program for logging into a remote machine and for executing commands in a remote machine. It is intended to replace rlogin and rsh, and provide secure encrypted communications between two untrusted hosts over an insecure network. X11 connections and arbitrary TCP/IP ports can also be forwarded over the secure channel.

OpenSSH is OpenBSD's rework of the last free version of SSH, bringing it up to date in terms of security and features, as well as removing all patented algorithms to separate libraries (OpenSSL).

GCC 3.0: The State of the Source

Mark Mitchell Alexander Samuel
CodeSourcery, LLC
mark@codesourcery.com samuel@codesourcery.com

August 24, 2000

1 Introduction

The GNU Compiler Collection (GCC) is the most fundamental component of the GNU/Linux developer's toolchest. GCC, like the Linux kernel and the X windowing system, is a complex but important part of the GNU/Linux operating system. In fact, both the kernel and X are built with GCC, so, to a large extent, the speed and correctness of the entire system depends on GCC.

The next major release of GCC, GCC 3.0, will be released sometime late this year. This release of GCC will contain a number of features of considerable import to the community. In addition, the quality assurance processes for this release will be more stringent than in any previous release. Therefore, GCC 3.0 will likely be a more reliable compiler, and will be more capable of supporting the needs of an ever-expanding developer community, than previous releases of GCC.

Principal among the improvements will be a new, stable, industry-standard C++ application binary interface (ABI). This new C++ ABI will provide, for the first time, an assurance that C++ programs compiled with one version of G++ can be linked with C++ libraries compiled with a different version of G++, and, in fact, with libraries compiled with other compilers.

In addition, GCC 3.0 will contain much improved support for Java, a number of new optimizations and bug fixes, improvements in compile-time performance, and new infrastructure to support future optimization and enhancement.

Because much of our work has focused on the C++ front-end, this paper will focus on C++-related work. That orientation should not be construed as implying that the other improvements in GCC 3.0 are less important. While a perhaps disproportionate amount of this paper focuses on the C++ ABI, we have attempted to describe, in somewhat less detail, some of the other highlights of the upcoming release.

2 General improvements

Significant efforts have been made during this release cycle to make GCC easier to maintain and improve in the future. Although these infrastructure investments do not have immediate user-visible benefits, their import cannot be overestimated. A well-organized, well-designed compiler means better compilers down the road.

2.1 Runtime support library

GCC provides certain support functions (like support for arithmetic on numbers with more bits than are supported on the target hardware, and support for exception-handling) in a special library, traditionally called `libgcc.a`. On non-GNU/Linux systems, this kind of compiler support functionality is typically provided in the system C library. That is a workable solution if the same vendor produces both the C library and the compiler. When GCC runs on non-GNU systems, however, there is no way to control the contents of the C library, and GCC must therefore provide its own runtime support library.

Unfortunately, `libgcc.a` contains some global data. It is important that there be only one copy of this global data in a complete program. However, in earlier versions of GCC, `libgcc.a` was linked into every shared library created by GCC. That could result in multiple copies of the global data, and incompatible support routines, in a complete executable, with the result that linking together shared libraries compiled with different versions of GCC did not always work.

In GCC 3.0, these problems will be solved by providing `libgcc` as a shared library. Libraries and executables will be linked against this shared library, which should prevent the kinds of incompatibilities described above.

2.2 Memory management

The memory management scheme used by the compiler itself was radically altered for the GCC 3.0 release. Memory allocated by the compiler is now garbage collected; previous releases used a complex system of memory pools. This change greatly reduced the number of memory-allocation bugs in the compiler, and simplified the implementation of new features.

Use of garbage collection, and other associated improvements associated with memory management, have dramatically reduced the memory footprint of the compiler in some cases. There have been improvements as great as 60% (from approximate 300 MB to approximately 100MB) when compiling some C++ programs.

2.3 Parsing whole functions

In early 2000, we converted the C++ front-end to produce a parse-tree for an entire function. (Previously, parse-trees were only available for individual statements.) G++ can now parse all of the code for a function before committing to the code shape for the function. This change has already lead to significant developments. The C++ front-end takes advantage of this representation to perform function inlining at a higher level, which allows more functions to be inlined with less memory usage. Future optimizations will take advantage of the fact that the parse tree for the complete function is available. For example, the "scatter-gather" optimization whereby a structure with several scalar members is treated simply as a collection of scalars can be implemented using the new representation.

Furthermore, the availability of a parsed representation of the entire program makes possible entirely new tools. A research group at Stanford is making use of the new representation to perform domain-specific error checking; they have, for example, detected bugs in the Linux kernel that could result in deadlocks. This analysis is only possible because the compiler is able to provide the error-checking tool with a representation of the entire program.

In addition, SGI was able to connect the G++ front-end to their IA64 back-end with relative ease by taking advantage of the new representation. They simply translated G++'s high-level representation for the program into a format understood by the SGI optimizers. In the future, it is likely that source browsers and other similar tools will also take advantage of these features. In fact, there has even been discussion of providing a plug-in interface that would allow the insertion of domain-specific optimization passes.

In the near future, we will convert the C front-end to use the same representation as the C++ front-end, which will yield all of the improvements listed above for C as well as C++.

2.4 Using the flow graph

Richard Henderson and others have made dramatic improvements to the optimization framework used by GCC. In particular, Richard's work allows GCC's optimization passes to access the flow graph for the function in a convenient way. (The flow graph shows where branches and loops can occur in the function.) Optimizations based on the flow graph are so-called "global" optimizations. The use of these techniques will allow the compiler to perform much more comprehensive optimizations than was previously possible.

GCC contains some experimental code to translate to and from single static assignment (SSA) form. SSA form is a means of representing the program that allows the compiler to perform many optimizations more easily. A compiler that takes advantage of SSA form can generate better code, and it can generate that code faster. In the future, we hope to convert many of GCC's high-level optimization passes to use this framework.

2.5 New optimizations

GCC 3.0 contains some powerful new optimizations. A new basic block reordering pass reorders generated code to improve cache performance based on either estimates of branch probabilities, or using output from profiled execution of the code. New exception-handling optimizations take advantage of throw specifications to eliminate unneeded exception handlers.

Perhaps most importantly, GCC 3.0 will contain a ground-up rewrite of the x86 back-end. Since so many GNU/Linux users run on x86, improvements in this area are clearly very important. The new back end describes the architecture much more accurately, and thereby allows GCC to generate considerably better code. There are also new optimizations targeting AMD's Athlon processor as well Intel's Pentium II and Pentium III processors.

2.6 Java

GCC 3.0 will include the GCJ Java compiler and the libgcj runtime support libraries. GCJ compiles Java code directly to native code just as the C, C++, and Fortran compilers do. In fact, GCJ uses the same object layout for Java objects that is used by GCC for C++ objects, so it is relatively easy to write programs that consist partly of C++ code and partly of Java code.

GCJ supports most Java language features present in version 1.1 of the JDK. As of yet, the library does not support many of the popular Java APIs available from Sun, but work is underway on the implementation of these APIs.

2.7 Bug tracking

The GCC project now makes use of the GNATS bug-tracking tool. That tool has made it a lot easier for users to report bug reports and for the GCC maintainers to respond to them. Geoff Keating set up an automated regression-testing tool that informs contributors when changes cause regressions in the GCC test suite. Increased reliance these automated tools will make it easier to make high-quality GCC releases.

3 C++ improvements

In recent releases of GCC, such as the GCC 2.95 release, GCC's support for modern C++ programming has increased dramatically. Our contributions have brought considerable improvements in ANSI/ISO conformance, increased the robustness of the compiler considerably, improved compile-time performance, introduced new optimizations, and improved error-reporting. The GCC 2.95 release was the first to compile complex expression-template programs, such as programs using the Blitz numerical programming library.

However, there has not been a stable C++ ABI. The ABI includes decisions made by the compiler as to how big objects will be, where data members will be located within an object, and the interfaces to run-time support functionality like exception-handling and run-time type identification. If the ABI changes between two versions of a compiler, then libraries created with the first version of the compiler cannot be linked with code compiled with the second version, and vice versa.

Frequent changes in the ABI therefore prevent people from distributing binary versions of C++ libraries. Distributors of free or open-source libraries suffer since it is harder for users to easily download the library. (Just because the source is available does not mean that users want to actually compile the source! The success of the various GNU/Linux distributions is proof of this fact; users are happy to be able to easily install the system without having to compile everything themselves.) Proprietary vendors suffer to an even greater degree; they do not wish to distribute source code, and they cannot easily provide object code that will work smoothly with various versions of the compiler.

The C++ ABI has changed frequently between previous GCC releases for a variety of reasons. To some extent, it has changed accidentally; changes to the compiler can result in inadvertent changes to the ABI. Until now, there have been no tests in the GCC test-suite that specifically test the C++ ABI, and for that reason it has been difficult to verify that changes to the compiler did not alter the ABI. The ABI has also changed out of necessity; as additional features mandated by the ANSI/ISO C++ standard have been implemented, ABI changes have been required in some cases.

The GCC maintainers have recognized the value in a stable C++ ABI for some time, and had in fact begun to work on implementing a new ABI. Fortuitously, a number of UNIX vendors and other interested parties joined forces to develop a C++ ABI for Intel's new IA-64 architecture. Their goal was to enable programs compiled with one compiler to link with libraries compiled with another compiler, or even to run a program compiled for one IA-64 operating system on another IA-64 operating system. (For example, a program compiled for the Monterey operating system being developed by IBM and SCO could be run on GNU/Linux, provided that the necessary support libraries on both systems adhere to the specified ABI.)

We participated in the effort to formulate the IA64 C++ ABI, and have now completed implementations of the ABI in G++ and in other compilers. Although the new ABI has only been formally specified for the IA-64 architecture, the ABI has been generalized to handle other architectures as well, since most of the design choices are equally applicable. So GCC 3.0, and subsequent releases of GCC, will use this same ABI on all architectures. The new ABI should greatly increase the ability of vendors to ship libraries that can be linked with versions of GCC that are released after the library was created.

In fairness, it is likely that some bugs remain in the GCC 3.0 ABI implementation. These bugs might necessitate minor changes in the future, but we expect that any such changes will not impact most programs, and should be resolved in the relatively near future.

3.1 Benefits of the new ABI

The new ABI has many benefits in addition to the stability that it will bring. In particular, the ABI committee worked hard to reduce the performance penalties long associated with some C++ features. Some programmers have avoided C++ altogether because of perceived performance problems. Others have formulated coding standards that prohibit the use of virtual functions, virtual base classes, or exception-handling in order to maximize performance. The committee took the implicit criticisms seriously, and attempted to design an ABI that would minimize the costs associated with these features. Many of the design decisions were based on existing practice in compilers available from EDG, HP, IBM, SGI, Sun and other vendor. Therefore, credit for these designs should be attributed not to the committee, but to the original designers of these optimizations.

Some of the changes in the new ABI are necessary simply to ensure correctness. For example, C++ has complex rules involving how virtual function calls should be dispatched during object construction and destruction. These rules could not be implemented using the G++ ABI on Linux, with the result that the GCC maintainers received frequent bug reports about which nothing could be done. The new ABI also contains a number of performance improvements, including dramatically shorter mangled names, reduced memory usage, and faster virtual function calls.

3.1.1 Mangled names

Many users reported mangled names of several kilobytes; in most cases the new ABI will reduce the length of these names to tens of characters. That change will reduce the size of object files, and make linking faster.

Much of the improvement stems from the observation that type signatures for template functions (which typically have the longest mangled names), often involve the same types. For example, the full signature for for `vector<vector<int> >` is:

```
vector<
  vector<int, std::allocator<int> >,
  std::allocator<std::vector<int,
                 std::allocator<int> > > >
::vector(void)
```

Here, the templates `vector` and `allocator` are repeated several times. If the element type was itself a template type, then there could be even more repetition.

Using the old ABI, the mangled name for this constructor is the following 173-character string:

```
__Q23stdt6vector2ZQ23stdt6vector2ZiZ
  Q23stdt9allocator1ZiZQ23stdt9allocator1Z
  Q23stdt6vector2ZiZQ23stdt9allocator1ZiRC
  Q23stdt9allocator1ZQ23stdt6vector2ZiZ
  Q23stdt9allocator1Zi
```

Using the new ABI, this same function is mangled as the more reasonable 39-character string:

```
_ZNSt6vectorIS_IiSaIiEESaIS1_EEC1ERKS2_
```

These compression techniques make for even greater savings when using more complex templates, such as those found in complex expression-template libraries. Part of the reason for the savings in this example is that the new mangling scheme also contains special abbreviations for some of the names in the standard library, including `std::string`. Since so many functions take parameters that involve these types, these abbreviations are very valuable.

3.1.2 Constructing virtual bases

The new ABI reduces the penalty for using virtual bases in several ways. Consider a constructor for a class that has virtual base classes. That constructor must decide whether or not to call the constructor for the virtual base subobject. The constructor for the virtual base class should not be called more than once for any single object. In the old ABI, the constructor took an additional parameter that told it whether or not to construct the virtual base classes. When a complete object was created, its constructor was called with the parameter set to a non-zero value. This value indicated that virtual base classes should be constructed. When the constructor for the complete object called base class constructors, the extra parameter was set to zero to indicate that constructors for virtual base classes should not be run again.

Unfortunately, setting up the parameters, and then checking their values, can take a substantial amount of time. The new ABI defines two different entry points for each constructor: one that constructs virtual bases, and one that does not. These may be alternate entry points into the same function, or they may be entirely separate functions; the choice is up to the compiler. Using alternate entry points, rather than a parameter, to distinguish between the constructors eliminates the run-time overhead of passing and checking the parameters.

3.1.3 Laying out virtual bases

As another example of the sorts of improvements provided by the new ABI, we consider the way in which virtual base classes are laid out. Virtual base classes present a problem in that their location, relative to the derived class that contains them, is not known when the program is compiled. For example, consider this fragment:

```
class V {
public:
  virtual void f();
  int i;
};

class D1 : virtual public V {
public:
  double d1;
};

void f (D1* d1) { d1->i = 3; }
```

The location of `i`, relative to `d1`, depends on the dynamic type of the object pointed to by `d1`. That this is the case stems from the fact that `V` is a virtual base; there is only one copy of `V` even in multiple classes in a hierarchy derived from `V`.

Traditionally, G++ (following the lead of the original AT&T implementation of C++), handled virtual bases by creating, in the derived class, a pointer to the virtual base. When `D1` is used as a base class, it is laid out like the following C structure:

```
struct D1 {
  double d1;
  V* vbase;
};
```

A complete object of type `D1` looks like:

```
struct D1_Complete {
  double d1;
  V* vbase;
  V v;
};
```

The v field is filled in with the address of the virtual base `v`. So, the expression `d1->i = 3` would have been implemented as `d1->vbase->i = 3`.

Now, consider what happens if several classes derived from V. For example, suppose we extend our example with:

```
class D2 : virtual public V { double d2; };
class D : public D1, public D2 {};
```

The layout for a complete object of type D would look like:

```
struct D_Complete {
  D1 d1base;
  D2 d2base;
  V vbase;
};
```

Note that both D1 and D2 contain pointers to V. Using this scheme, a complex hierarchy with many virtual bases, a large percentage of the space consumed by an object can end up devoted to pointers to virtual bases. The old ABI squandered not only space (because of all the pointers), but also time: initializing all of the pointers makes object construction and destruction unnecessarily costly.

In practice, hierarchies making use of virtual bases almost always make use of virtual functions. Therefore, each object already contains a "vtable pointer", i.e., a pointer to a virtual function table. Dynamic dispatch is implemented using the virtual function table; when the *i*th virtual function in a class is called, control is transferred to the address given by the *i*th entry in the virtual function table. In addition to the virtual function addresses, the new ABI stores offsets from derived classes to their virtual bases in the virtual function table. This mechanism eliminates the need to store pointers to virtual bases in the objects themselves, thereby eliminating both the space penalty, and the time penalty for initializing the objects.

3.2 C++ standard library

The implementation of the C++ standard library that shipped with GCC 2.95.2 and previous releases was woefully out-of-date. It provided no support for wide characters or locales, did not place names in the `std` namespace, did not support templatized I/O streams, and contained numerous other deficiencies.

Benjamin Kosnik and others have worked very hard over the last several years on a ground-up rewrite of the library. GCC 3.0 will be the first GCC release to use the new standard library, which is now mostly complete. Already, the deficiencies mentioned above have been repaired, and users will find that the new library conforms much more closely to the ANSI/ISO C++ standard.

3.3 C/C++ preprocessor

GCC 3.0 will include a new implementation of the C preprocessor, contributed by Zack Weinberg and others. The new preprocessor is faster than the old preprocessor. More importantly, the new preprocessor will allow direct integration with the compiler front-ends themselves. Presently, GCC first creates a temporary file containing the preprocessed source file. The front-ends then process this preprocessed file. The overhead of writing, and then reading, the temporary file is considerable.

In addition, the process of tokenization is needlessly duplicated between the preprocessor and the compiler front-ends. For example, when confronted with the string `f (71)` the preprocessor determines that the token stream consists of an identifier, an opening parenthesis, a numeric literal, and a closing parenthesis. The compiler must perform the same analysis.

The new preprocessor will be able to connect directly to the compiler front-ends, without using an intermediate file. In this way, the overhead of reading and writing the file will be eliminated, as will the redundant retokenization. In this way, the compile-time performance of GCC should be improved substantially. Initial measurements show as much as a 10% speedup when compiling some programs with the new preprocessor.

4 Future directions

The GCC community has already started talking about what will happen after GCC 3.0. In addition to the usual improvements, we are hoping to incorporate two new languages into the GNU Compiler Collection: Pascal and ADA. Unifying the development groups that have been working on these front-ends with the core GCC development time should make it much easier for users of Pascal and ADA to obtain compilers that interoperate smoothly with the rest of GCC. On the other hand, since there has been little interest in Chill, it is possible that the Chill front-end will be dropped from GCC at some point in the future. The C and C++ front-ends may be combined into a single front-end in order to reduce code duplication and to make it easier to achieve consistent semantics between C and C++.

There have been serious discussions about making relatively major changes in the internal data structures used by GCC in order to offer new optimization opportunities. Presently, GCC uses two major representations of the source program: an abstract syntax tree representation that is close to the source language, and a register transfer language that is close to the eventual generated machine code. Unfortunately, there is no convenient intermediate representation: one which is simpler than the source language, but that still abstracts away from the machine representation.

This "representation gap" makes it difficult to implement many high-level optimizations in GCC. For example, it is difficult to implement many loop optimizations. Some of the algorithms that are implemented in GCC suffer both in implementation complexity and in the quality of the generated code by having to work on inconvenient representations of the program. Therefore, it is likely that structural changes will be made to accommodate a new representation.

5 Challenges

GCC has a lot going for it. It is a free, easily retargetable optimizing compiler supporting the most important available compiled programming languages. Unfortunately, there are some notable weaknesses in GCC as well. Presently, GCC's optimization is not as good as that provided by some commercial compilers. The error messages produced by GCC are not as helpful as they should be. Despite major improvements, there are still weaknesses in support for some language features in C++, Java, and Fortran. The compile-time performance of the compiler is not as good as that of the best commercial compilers. The documentation provided is not as good as it could be. There are substantial weaknesses in the supporting tools, including debuggers and profilers. Other important tools, like incremental linkers, source browsers, and integrated development environments, do not exist, are not of commercial quality, or not are not freely available. Quality assurance between releases has not been sufficient to ensure that users can easily upgrade from one version of the compiler to the next.

For these reasons, GCC will face threats in the future from commercial compilers. The success of GNU/Linux represents a new market for tools vendors; already, for example, there are two proprietary compiler products available for GNU/Linux. If these proprietary products are better than GCC, then many developers will use them. Many companies will find the cost of these products worth the price if they feel that these tools will enable faster, easier development.

Therefore, the community should continue to invest in GCC, by donating development effort, by providing hardware resources, and by funding future improvements in GCC. That up-front investment will ensure that GCC continues to improve, and that it remain the best available compiler for the GNU/Linux operating system.

SMP Scalability Comparisons of Linux® Kernels 2.2.14 and 2.3.99

Ray Bryant raybry@us.ibm.com Bill Hartner bhartner@us.ibm.com
Qi He[1] qhe@cc.gatech.edu Ganesh Venkitachalam[2] venkitac@yahoo.com
IBM® Linux Technology Center
IBM Austin

Abstract

The Linux 2.4 kernel should provide significantly better SMP scalability than is available in the 2.2 series. While the development of the 2.4 kernel is still continuing, performance studies on the 2.3.99 kernels can be used as an indicator of what the performance of the 2.4 kernel will be like. In this paper, we compare performance and SMP scalability of Linux 2.2.14 and Linux 2.3.99 on the Intel® 32-bit platform using four benchmarks. Three of these are targeted to specific system components; the fourth is SPECweb99™. SMP scaling for these benchmarks is shown to be significantly better with 2.3.99 than it is with 2.2.14. This should translate into significantly better SMP scaling (and hence additional raw performance on SMP systems) for applications running under Linux 2.4.

Introduction

Just as other authors have compared the function available in Linux 2.4 to that available in Linux 2.2 (see, for example, [Linux 2.4]), the purpose of this paper is to compare the performance likely to be available in Linux 2.4 to the performance delivered by Linux 2.2. We make this estimate by comparing the performance of Linux 2.3.99 kernels to Linux 2.2.14 on the Intel 32-bit platform using a set of four benchmarks. Three of these benchmarks are targeted to specific components of the Linux kernel:

- Volanomark™: a benchmark of a chat room server written in the Java™ language. (scheduler, TCP/IP stack)
- Netperf: a network communications benchmark (TCP/IP stack)
- FSCache: a file system buffer-cache benchmark (file system cache)

The fourth benchmark, SPECweb99 [SPW99] is a benchmark of web server performance.

In addition to comparing the performance of these kernels, we use a standard set of performance tools to analyze the kernels as the benchmarks execute. The resulting data will help us analyze and fix bottlenecks remaining in Linux 2.4.

While benchmarking is of course, imperfect (if not controversial, c. f. [Mindcraft] [ESRFiasco]), we believe this is the only way to obtain reproducible and verifiable comparisons between two kernels. Such a study should be conducted so that another group running the same benchmark with the identical setup will obtain similar results. One must also be careful not to infer conclusions from the benchmark results about unrelated workloads.

In the following sections of this paper, we discuss the following:

- the instrumentation patch we used to analyze kernel performance
- the kernels we compared
- benchmarks employed
- the measurement environment
- the measurement results
- implications of these results pertaining to the performance of the Linux 2.4 kernel

Instrumentation

This paper presents results obtained from the IBM Linux Kernel Trace and Profile Facility patch.[3] This patch implements a profiling facility that profiles the

[1] Author's current address: Georgia Institute of Technology, 328290, Georgia Tech Station, Atlanta, GA 30332.
[2] Author's current address: VMware, Inc., 3145 Porter Drive, Palo Alto, CA 94304.
[3] This code is not available outside of IBM at the present time. If there is sufficient community interest, it may be released to the open source community at some time in the future.

kernel execution. It produces output that is similar to the output produced by the Linux kernel profiling facility or the SGI™ Kernprof [SGIKernprof]. The measurements reported using the IBM Kernel Trace and Profile Facility could, in principle, be duplicated outside of IBM using these other tools.

In addition to timer-based profiling, both the IBM Linux Kernel Trace Facility and the SGI Kernprof Patch support Pentium® performance-counter based profiling. Time-based profiling samples the current instruction pointer at a given time interval (e.g., every 10 ms.). Postprocessing tools use the recorded locations to construct a histogram of the amount of time spent in various kernel routines. In performance-counter based profiling, a profile observation is set to occur after a certain number of Pentium performance counter events [PerfCount]. For example, one could take an observation every 1,000,000 instructions or every 10,000 cache-line misses. Just as time-based profiling shows where the kernel spends its time, an instruction-based profile shows where the kernel executes most of its instructions and a cache-line based profile shows where the kernel takes most of its cache-line misses. These latter types of profiles can provide additional insight into kernel performance.

Kernels Compared

In this paper, we report on comparisons of Linux kernel 2.2.14 and Linux 2.3.99-pre4, pre6, or pre8 as specified in the benchmark data. For some of the benchmarks the kernel.org version of 2.2.14 would not correctly run the benchmark and we had to use the Red Hat™ 6.2 version of the kernel instead (2.2.14-5.0). We were later able to fix this problem by porting the eepro100 ethernet card driver from Red Hat 6.2 to the kernel.org version. The versions of the 2.2.14 kernel used for the SPECweb99 experiments were the kernel.org version of 2.2.14 with the updated version of the eepro100 driver.

The kernels were all built in uniprocessor and multiprocessor versions using gcc version "gcc version egcs-2.91.66 19990314/linux." Kernels were built for machine architecture "686". In some cases we also include measurement data from other kernel versions for comparison purposes. These kernels were also built using the same version of gcc.

A full specification of the configuration options used to build these kernels would be required to completely define the way these kernels were built; these configuration files are available from the authors on request.

Benchmarks Used

Volanomark

Volanomark is a benchmark of a chat room server and is written in the Java language. The benchmark and the chat room server are products of Volano, LLC [Volano]. Occasionally, Volano publishes a report (known as the Volano Report [VReport]) comparing Volanomark performance on a variety of Java implementations and operating systems. The Volanomark measurements reported in this paper use slightly different run rules and parameters than the measurements reported in the Volano Report and are not comparable to the results published there.

We present throughput results using Volanomark 2.1.2 and the IBM Runtime Environment for Linux version 1.1.8 (formally known as part of the IBM® Developer Kit for Linux®, Java™ Technology Edition, Version 1.1.8, and herein referred to as the IBM R/T 1.1.8) that we used in a previous paper [JTThreads]. Further description of Volanomark can be found at [VMark] and in our previous paper. We also present SMP scalability results using Volanomark 2.1.2 and the IBM Runtime Environment for Linux version 1.3 [NewIBMRT] (formally known as part of the IBM® Developer Kit for Linux®, Java™ 2 Technology Edition, Version 1.3, and herein referred to as the IBM R/T 1.3) and IBM R/T 1.1.8.

The principle metric reported by the Volanomark test is "chat server message throughput" in messages per second. All Volanomark results present here are for loopback experiments; in a loopback experiment both the client and server run on the same system.

While Volanomark was developed by Volano, LLC to compare their chat server performance under different Java implementations, we have found it to be useful in the Linux environment to test scheduler and TCP/IP stack performance, particularly if the IBM R/T 1.1.8 or 1.3 is used. Each chat room client causes 4 Java threads to be created. The IBM R/Ts use a separate Linux process to implement each java thread. Thus for a Volanomark run with 200 simulated chat room clients, there will be 800 processes active in the system. Each java thread spends most of its time waiting for data from a communications connection. When data is received, relatively little user space processing is performed, new messages are sent, and the Java thread then waits for more data. The result is that approximately 60% of the time spent executing the benchmark is in kernel mode.

Netperf

Netperf is a communications benchmark available from [NetPerf]. Netperf includes a number of tests that can be used to measure protocol and network performance.

In this paper we report on the "request response" or "RR" test. The RR test creates a number of socket connections between the client and server. Once the connections are established, messages are transmitted and returned across each connection as quickly as possible. At the end of a fixed time period, the test ends and the clients report the total number of messages exchanged. The principle metric produced by the Netperf RR test is message throughput in messages/second.

FSCache

FSCache is an internal IBM benchmark that measures file system performance for files read out of the buffer cache. (It is our plan to make this benchmark available to the open source community.) FSCache supports both random and sequential read tests; here we report only the results of the random read tests. The principle metric of FSCache is kilobytes/second read from the buffer cache.

This benchmark represents only one aspect of measuring file system performance. However, it is necessary that this component of the file system scale in order for the overall file system to scale well. Thus our results are preliminary indications of overall file-system scalability.

SPECweb99

SPECweb99 was developed by the Standard Performance Evaluation Corporation (SPEC), and is described in detail at [SPWB99]. The version of SPECweb99 we use is the 1.01 version released on 11/23/99. SPECweb99 is designed to test and measure the ability of a particular system and hardware configuration to act as a web server, delivering both static and dynamic content pages to a network of client systems driving the workload. The benchmark measures the number of simultaneous client connections that a server is able to support, while maintaining predefined bit rate and error rate limits. A connection that meets the bit rate and error limits is said to be "compliant"; the primary statistic of the SPECweb99 benchmark is the number of compliant connections.

The benchmark has been designed to favor a web server that is capable of supporting a larger number of relatively slow connections over a web server that is capable of supporting a smaller number of relatively fast connections. The former situation is regarded as being more representative of the web server environment on the Internet and is one of the improvements made to this benchmark from its predecessor (SPECweb96™).

Another improvement in SPECweb99 over SPECweb96 is the inclusion of dynamic as well as static content. Both dynamic GET and POST operations are performed as part of the SPECweb99 workload. The dynamic GETs simulate the common practice of "rotating" advertisements on a web page. The POSTs simulate entry of user data into a log file on the server, such as might happen during a user registration sequence. Dynamic content comprises 30% of the workload and the remainder is static GETs. The dynamic workload is a mixture of POSTs, GETs, GETs with cookies, and a small fraction is due to CGI GETs. The proportions were based on analysis of workloads of a number of internet web sites.

The file access pattern is also modeled after the access patterns found on a number of internet web sites. Files are divided into a number of classes and a Zipf distribution is used to choose files to access within each class. The total number of bytes accessed increases proportionally to the number of connections to the web server. Further details on the file access and HTTP request type distributions can be found in the SPECweb99 FAQ [SPWBFAQ].

The web server software used is independent of the SPECweb99 benchmark. For our test configurations, we used the Zeus 3.3.5a [Zeus] web server with the tuning suggestions as provided by SPEC at [SPTune]. The dynamic content implementation for the Zeus web server was also obtained from the SPECweb99 site [SPWB99]. This implementation is entirely in the C programming language.

Presentation of SPECweb99 Results

The SPECweb99 benchmark is a licensed benchmark of the SPEC organization and its results can only be used and reported in certain ways defined by the license. In particular, the SPECweb99 statistic can only be reported for a particular system after the result has been submitted to and approved by the SPECweb99 committee. In this paper, it is not our primary goal to report the SPECweb99 statistic, rather, we are interested in using this workload as a basis for comparing the 2.2.14 and 2.3.99 Linux kernels.

Our use of SPECweb99 thus falls into the category of "research" use. At the time of the writing of this paper, the SPECweb99 license did not include a research use clause, although other SPEC benchmarks do include such a clause. Our results are therefore presented under a special agreement with SPEC [SPNote]; we expect that a research usage clause will become part of the license for the SPECweb99 1.02 release.

The terms of the agreement are as follows:

- We agree to provide full disclosure of the details of the benchmark execution. Any details not otherwise provided in this paper can be obtained from the authors.
- Make compliant runs. That is, all benchmark and workload rules will be adhered to.
- The dynamic content implementation will be one from the SPECweb99 site and is thus an implementation that has been audited for conformance to the benchmark rules by the SPECweb99 committee.
- Only relative results will be reported. In our case all results will be scaled by the result for the 2.2.14 UP kernel at the lowest number of connections tested.
- We will not make use of the SPECweb99 statistic (number of compliant connections).

Although we do not report the SPECweb99 statistic, we do report values for the following quantities, as measured by the SPECweb99 client software:

- Average bitrate/second per connection: The average number of bits delivered per connection in Kb/s.
- Average Latency : the average response time per operation in milliseconds.
- Operations/second: The average number of HTTP operations performed across all connections in operations per second.

Here the averages are performed across all connections and throughout the duration of the measurement run as defined by SPECweb99. As specified by the SPECweb99 rules, each run is performed 3 times and the median result of the runs is reported.

Measurement Environment and Experiment Run Rules

Measurements in this paper for all tests except the Volanomark scalability tests are reported for IBM Netfinity® 7000 M10 systems. These are Intel® Pentium® II Xeon™ 4-way SMP systems. For the Volanomark throughput, Netperf, and FSCache tests, the system being measured was a 400 MHZ system with 1.5 GB of RAM. DASD on this system was accessed using an Adaptec® 7858 SCSI driver. For the SPECweb99 test, the system measured was a 450 MHZ system with 4GB of RAM. System data on this system was accessed using an Adaptec 7858 SCSI driver; the SPECweb99 data is accessed via an IBM ServRAID™ device spanning 9 physical disk drives.

The NIC cards in the Netfinity systems are IBM 10/100 EtherJet™ PCI cards; the Intel 8-way uses an Intel Pro 100 ethernet PCI card. In all cases the device driver was the eepro100 driver.

Volanomark Setup

For the Volanomark throughput tests, the server employed was a 400 MHZ Netfinity system. For the Volanomark scalability tests, the server was an Intel Pentium III Xeon 8-way operating at 550 MHZ. The systems were booted with 1GB of RAM and for the N-way speedup test the system was booted with N processors.

Netperf Network and Client Setup

For these tests the client machine was the 8-way Intel Pentium III 550 MHZ system. This client was connected to the server system using a single, 100 MB Ethernet operating in full duplex mode. *vmstat* was run on the server to measure processor utilization. A network sniffer was used to sample Ethernet utilization to make sure it was not becoming saturated and hence a bottleneck. The server machine was the 400 MHZ Netfinity 7000 M10 system.

FSCache Setup

The FSCache test was executed on a 400 MHZ Pentium II Netfinity system. In order to make sure that our measurements report scalability of the buffer cache and are not limited by the bandwidth of the underlying memory system, we first performed a series of experiments with file-system reads replaced by memory-to-memory copies. The reason for choosing this baseline experiment is that the time required to complete a read operation out of the buffer cache is typically dominated by the amount of time required to copy data from the kernel buffer cache to the user-space buffer. If this memory-to-memory copy operation does not scale then the corresponding FSCache benchmark may not scale. These experiments indicated that for file sizes larger than 1MB on the 400 MHZ Pentium II Netfinity system, the FSCache tests would not scale. Additionally when we ran the full FSCache tests, we found that scalability increased if we used file sizes of 128 KB. For this reason, we chose to use the (relatively small) file size of 128KB in our FSCache tests.

SPECweb99 Network and Client Setup

We are currently using a set of 20 clients and four 100 MB Ethernets to run our SPECweb99 experiments. The clients are connected to the server using a 24-port switch (20 ports for the clients and 4 ports for the server).

The network clients are 166-200 MHZ Pentium Pro machines running Microsoft® NT Workstation 4.0 with Service Pack 4. The NT Performance Monitor is used to ensure that client machines do not become the bottleneck. A network sniffer was used to sample network utilization and to ensure that the network does not become a bottleneck for these tests. To balance the workload across the client machines, the benchmark is configured so that the faster clients submit 1.5 times as many requests as the slower clients.

The server machine is a Netfinity 7000 M10 450 MHZ Pentium II 4 CPUs with 4 IBM EtherJet 10/100 ethernet cards and 4 GB of RAM. Since 2.2.14 does not support more than 2GB of RAM, all the experiments for 2.2.14 and most of the experiments for 2.3.99-pre8 were performed with 2GB of real memory; one data point for the 2.3.99-pre8 SMP case was run with 4GB of RAM.

Khttpd was not enabled for any of the runs reported here.

Statistical Properties of the Test Results

In this paper, numbers reported are based on the average of a number of repeated, identical, independent trials. The trials are done without rebooting the system; however the effect of the previous trial on the system is removed as much as possible before the next trial is started. Where there is a warm up effect that results in the first trial being different than the other trials, the results of the first trial are discarded.

Trials are repeated until the tests have "converged". This is stated as follows: "The test converged to an x% confidence interval width at y% confidence." What this means is that the results of a confidence interval estimated based on a Students-T distribution has a width of less than x% of the mean with a y% level of confidence.

After each trial (except for Volanomark), the length of the confidence interval is calculated, and if the length is small enough, the test completes and the average over the trials is reported as the result. In some cases, the trial does not converge after the maximum allowed number of iterations. In these cases, the test is re-run to obtain convergence. For Volanomark, a fixed number of trials is done, and convergence is tested after all trials are completed.

SPECweb99 comes with its own set of run rules and run acceptance criteria [SPWBFAQ]. The results of SPECweb99 reported here are run under the preprogrammed run rules of the test.

Measurements reported here as UP are for a uniprocessor kernel. Measurements reported as 1P are for multiprocessor kernels booted on a single processor. Comparisons between the 1P and UP measurements are particularly useful for evaluating the overhead of SMP synchronization and locking.

For all of the benchmarks discussed here, we define scalability as the ratio of the benchmark statistic for an SMP system to the corresponding benchmark statistic for a UP system. While this results in lower scalability numbers than one might get by dividing the SMP result by the 1P result (since the UP ratio penalizes the SMP system for locking overhead) we regard this as the fairest way to define scalability.

Measurement Results

Volanomark

In Figure 1 we show measurements of the throughput in messages per second for Volanomark with the IBM R/T 1.1.8 and 4 different Linux Kernels. In all cases the kernels are UP. We note that among the kernels shown, Linux 2.3.99-pre4 has the best performance. This is a good indication that Linux 2.4 should outperform Linux 2.2.14 on workloads similar to Volanomark.

In Figure 2 we show the results of a kernel profile measurement of the Linux 2.3.99-pre4 kernel while it is running Volanomark. As previously reported [JTThreads], this profile shows that the largest amount of system time is spent in the scheduler. While Volanomark is admittedly a stress test for the scheduler, it appears that additional enhancements will need to be added to Linux scheduler for workloads with large numbers of threads.

In Figure 3 we show scalability results for Volanomark when run under IBM R/T 1.1.8 and IBM R/T 1.3 for Linux kernels 2.2.14 and 2.3.99-pre4. This figure shows that while running Volanomark, the IBM R/T 1.3 scales better than IBM R/T 1.1.8, and that Linux 2.3.99-pre4 scales better than Linux 2.2.14. Of course, the speedup numbers obtained here are application and environment dependent and should not be taken as general statements of the speedup results an arbitrary workload might achieve while running under the IBM R/T 1.3.

Netperf

In Figure 4, we show the results of a Netperf comparison test between 2.2.14-5.0 (the Red Hat 6.2 Kernel) and 2.3.99-pre8. The horizontal axis represents the number of client threads used to drive the server; each client thread creates one connection to the server. The vertical axis represents the number of Netperf messages sent per second during the test. The four lines on the chart represent data for 2.2.14-5.0 UP and SMP and 2.3.99-pre8 UP and SMP. Note that the benchmark running on the Linux 2.2.14.-5.0 SMP kernel with 4 processors runs more slowly than the benchmark running on the Linux 2.2.14-5.0 UP kernel. Performance of the 2.3.99-pre8 UP kernel is better than the performance of *both* the UP and SMP trials for Linux 2.2.14-5.0. The Linux 2.3.99-pre8 SMP kernel performs dramatically better than all the other kernels and gives SMP scalability of 2.1 on this 4-processor system. Comparing the 2.3.99-pre8 SMP result at 40 connections to the 2.2.14 SMP result at 16 connections (these are the maximum throughputs for each case) shows that the 2.3.99 SMP kernel is able to deliver 3.1x times as many messages as the 2.2.14 SMP kernel on this benchmark.

In Figure 5, we show the results of time-based and instruction-based profiles of the Linux 2.3.99-pre8 kernel while running the Netperf benchmark. The time-based profile shows that the scheduler is again the kernel routine where the most time is spent during the benchmark. Given the very small message sizes used (4 bytes), this is not surprising. The differences between the two profiles indicate that in some routines (scheduler and stext) more instructions are executed per unit time whereas for other routines (speedo_start_xmit) relatively fewer instructions are executed per unit time. These are examples of the kind of differences one can see using a performance-counter event-based profile versus a timer-based profile.

FSCache

In Figure 6, we show FSCache scalability results for Linux 2.2.14 and 2.3.99-pre6. (The initial drop in scalability when going from the UP case to the 1P case reflects the overhead of SMP locking.) For the 2.2.14 case, the size of the buffer used had relatively little effect on the results so we only show the 4096 byte buffer case. For 2.3.99, the size of the buffer did make a significant difference; we show the results for 512 and 4096 byte buffers. As can be seen from Figure 6, Linux 2.2.14 provides no significant performance by adding additional processors for this benchmark, while we see that Linux 2.3.99-pre6 provides as much as a 2.5x increase on this 4-processor system.

SPECweb99

In Figures 7 through 12, we show the results of our SPECweb99 experiments. Figure 7 shows that the bitrate per connection falls off rapidly as the number of connections is increased for all cases except 2.3.99 SMP where only a slight decrease is detected. Similarly, Figure 8 shows that the response latency increases for all cases except 2.3.99 SMP where only a slight increase is detected. In Figure 9, we see that the operations/second handled by the server increases linearly (with the offered load) for the 2.3.99 SMP case and for all other cases it does not. Finally, in Figure 10 we see an underlying reason for these results. In all cases except 2.3.99 SMP, the system is CPU bound and cannot support additional work.

A plausible question, when comparing the CPU utilization curves for 2.2.14 SMP and 2.3.99 SMP is "Where did all of the extra CPU time come from for 2.3.99?". In Figures 11 and 12, we answer this question using a time-based profile of the kernel while it is running the benchmark. In Figure 11 we see that about 50% of the CPU-time consumed by the kernel was spent in stext_lock. (stext_lock appears in the profile when the kernel is spinning on a spinlock, so about 50% of the time was unavailable to service web requests). Figure 12 shows that less than 4% of the time was spent spinning for locks while running the benchmark under 2.3.99 SMP.

For Figures 7 through 10, the curves drawn represent data for 2GB of RAM. We also include a single data point for 2.3.99-pre8 and 4GB of RAM at 820 connections. This point indicates that the 2.3.99 SMP experiment at 820 connections and 2GB RAM may be memory bound. In Figure 9, the 4 GB data point appears to follow the linear trend established by the 420 through 660 connection data points. Similarly, if we examine the CPU utilization graph in Figure 10, it is apparent that with 4 GB of memory, the system is more fully utilized. If we compare operations per second achieved under 2.2.14 SMP at 500 connections and 2.3.99 SMP at 820 connections and 4 GB RAM (see Figure 9), we see that the latter kernel is able to deliver 1.85 as many operations per second as the former.

At 820 connections, our client network is fully utilized. Thus we are unable to state with certainty that we have reached the peak of the throughput curve for 2.3.99 SMP at 820 connections. We are in the process of obtaining a Gigabit ethernet switch in order to continue these experiments and find a final scaling number for 2.3.99 under this benchmark.

Concluding Remarks

We have presented measurements that show that the SMP scalability of the Linux 2.4 kernel should be significantly better than that of the Linux 2.2.14 kernel. In particular it appears that:

- Linux 2.4 performs better than Linux 2.2.14 for the Volanomark benchmark when run with IBM R/T 1.1.8.
- It remains to be seen whether or not the Linux 2.4 scheduler will still require additional scalability tuning for workloads that require many running threads.
- The scalability of the TCP/IP stack for a multiprocessor Linux 2.4 system is dramatically improved over the scalability of the TCP/IP stack for 2.2.14. This improvement alone should make the Linux 2.4 kernel perform substantially better under benchmarks such as the ones performed by Mindcraft [Mindcraft]. Our Netperf experiments showed that an SMP scalability of 2.1 out of 4 is possible with Linux 2.3.99-pre6. This benchmark performs 3.1 times better under 2.3.99-pre6 SMP than it does on 2.2.14 SMP.
- The SMP scalability of file system buffer cache access has been significantly improved. Our FSCache experiments demonstrated that a scalability of 2.5 out of 4 is possible with Linux 2.3.99-pre6.
- Our SPECweb99 experiments indicate that 2.3.99-pre8 SMP (running the Zeus web server) is able to deliver at least 1.85 times as many operations per second than 2.2.14 SMP. However, we do not yet have a final scalability number for 2.3.99 due to limitations of our current client network.

While these results indicate that Linux 2.4 should provide improved SMP scalability over that available in the current production kernels, much additional work remains before Linux can effectively exploit SMP systems larger than 4-way.

Acknowledgments

Jerry Burke, Scottie M. Brown and George Tracy of IBM were instrumental in setting up the SPECweb99 benchmark and we greatly appreciate their assistance. We also acknowledge the SPEC organization (Particularly Kaivalya Dixit and Paula Smith) for its permission to use the SPECweb99 benchmark in this paper.

References

[Linux2.4]: Wonderful World of Linux 2.4 (Final Draft), Joe Pranevich, http://linuxtoday.com

/news_story.php3?Ltsn=2000-07-17-014-04-NW-LF-KN

[Mindcraft]: Open Benchmark: Windows NT Server 4.0 and Linux, Bruce Weiner, http://www.mindcraft.com/whitepapers/openbench1.html

[ESR Fiasco]:ESR and the Mindcraft Fiasco http://www.slashdot.org/features/99/04/23/1316228.shtml

[SGI Lockmeter]:Kernel Spinlock Metering for Linux, http://oss.sgi.com/projects/lockmeter

[SGI Kernprof]: Kernel Profiling, http://oss.sgi.com/projects/kernprof

[PerfCount]: Intel Architecture Software Developer's Manual Volume 3: System Programming, http://developer.intel.com/design/pentiumii/manuals/243192.htm

[JTThreads]: Java technology, threads, and scheduling in Linux--Patching the kernel scheduler for better Java performance, Ray Bryant, Bill Hartner, IBM, http://www-4.ibm.com/software/developer/library/java2/index.html

[Volano]: Volano Java Chat Room, Volano LLC http://www.volano.com

[VReport]: VolanoMark Report page, Volano LLC, http://www.volano.com/report.html

[VMark]: Volano Java benchmark, Volano LLC, http://www.volano.com/benchmarks.html

[NetPerf]: Network Benchmarking NetPerf, http://www.netperf.org

[SPWB99]: Web Server benchmarking SPECweb99, Standard Performance Evaluation Corporation, http://www.spec.org/osg/web99

[SPWBFAQ]: SPECweb99 FAQ, Standard Performance Evaluation Corporation, http://www.spec.org/osg/web99/docs/faq.html

[Zeus]: Zeus WebServer, Zeus Technology, http://www.zeustech.net

[SPTune]: SPECweb99 Tuning Description, Standard Performance Evaluation Corporation, http://www.spec.org /osg/web99/tunings

[SPNote]: E-mail communication, Kaivalya Dixit, President of SPEC, the Standard Performance Evaluation Corporation, and Paula Smith, Chair of the SPECweb99 committee, 7/10/2000.

Trademark and Copyright Information

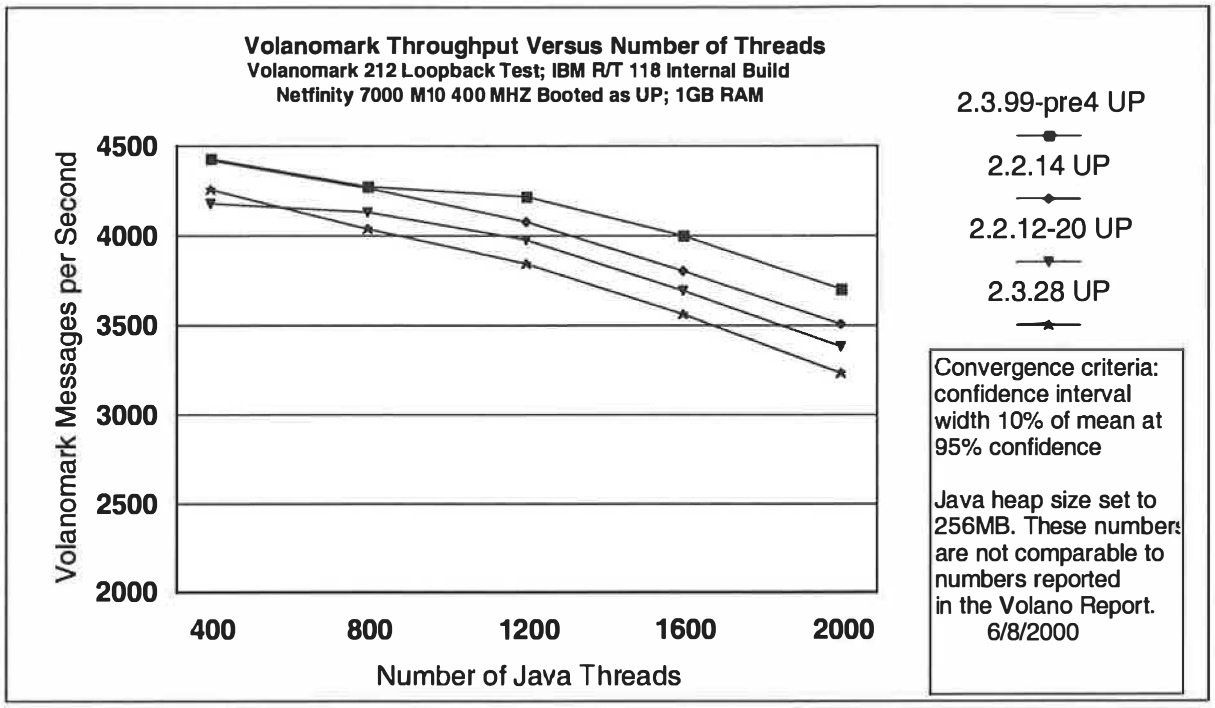


Figure 1

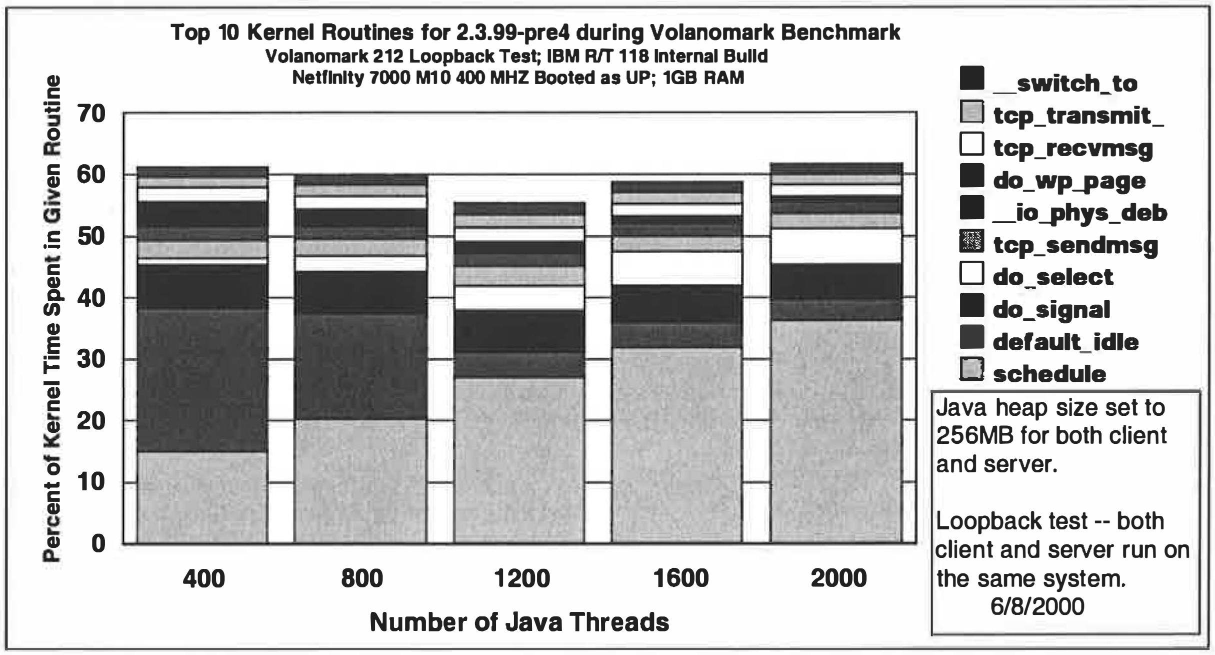


Figure 2

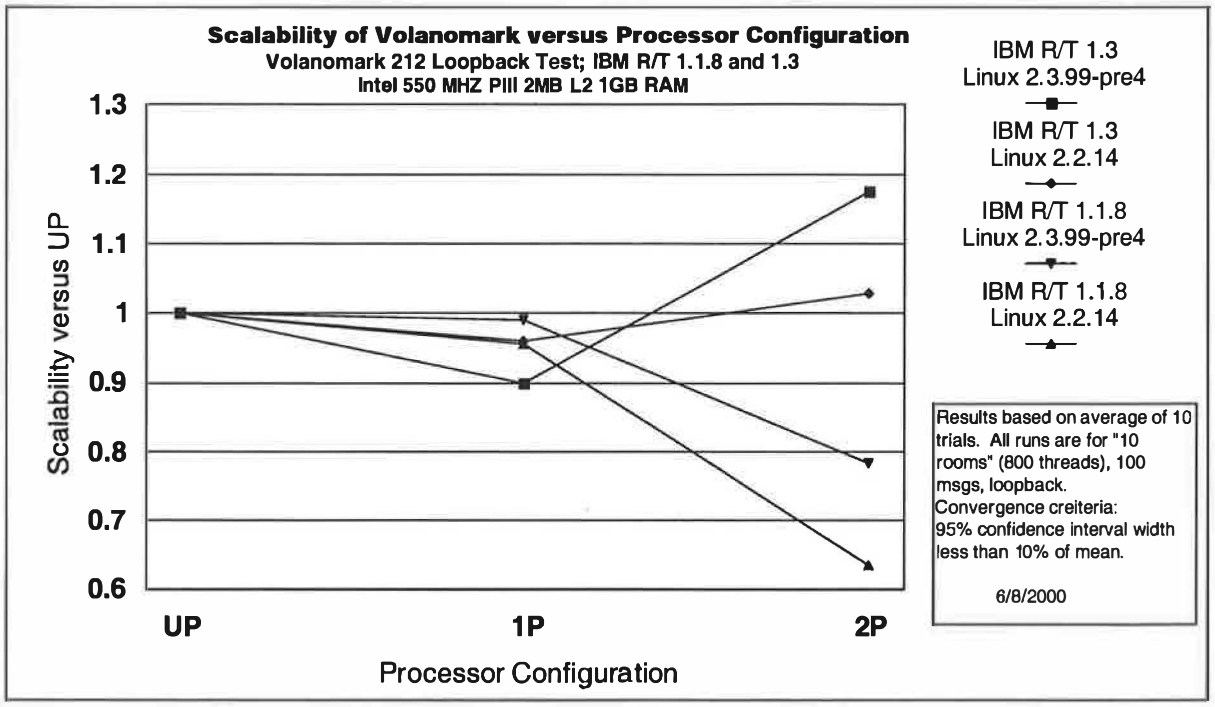


Figure 3

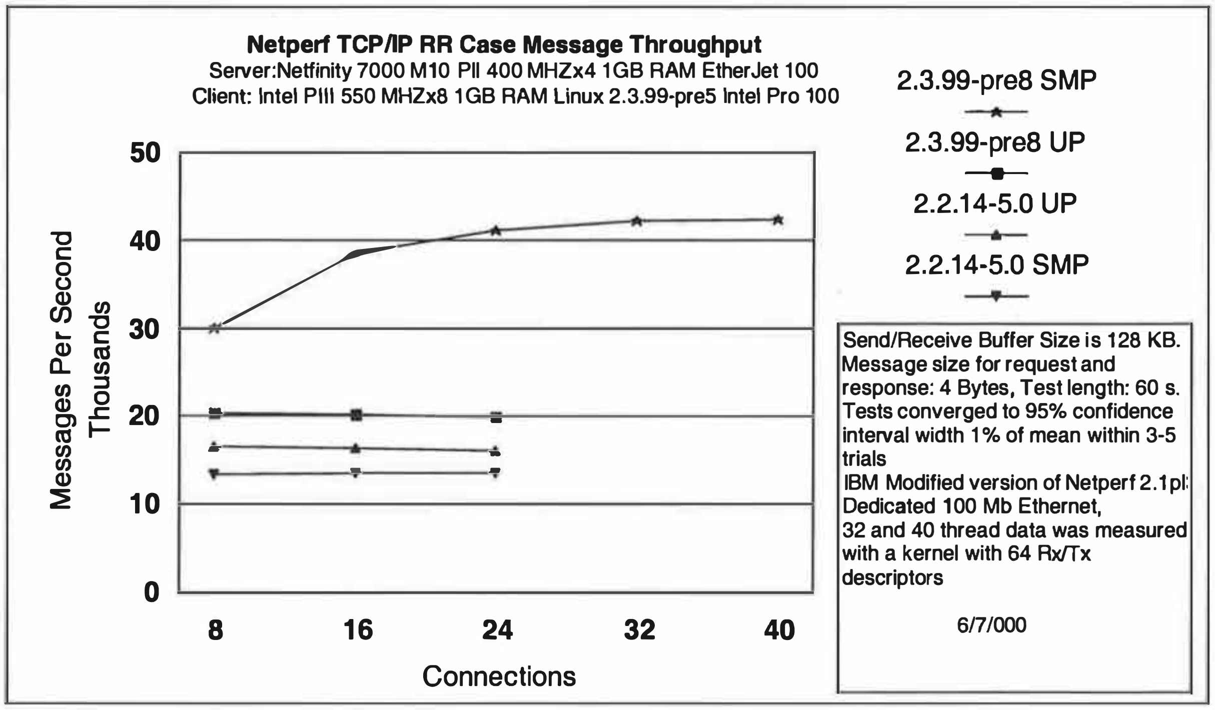


Figure 4

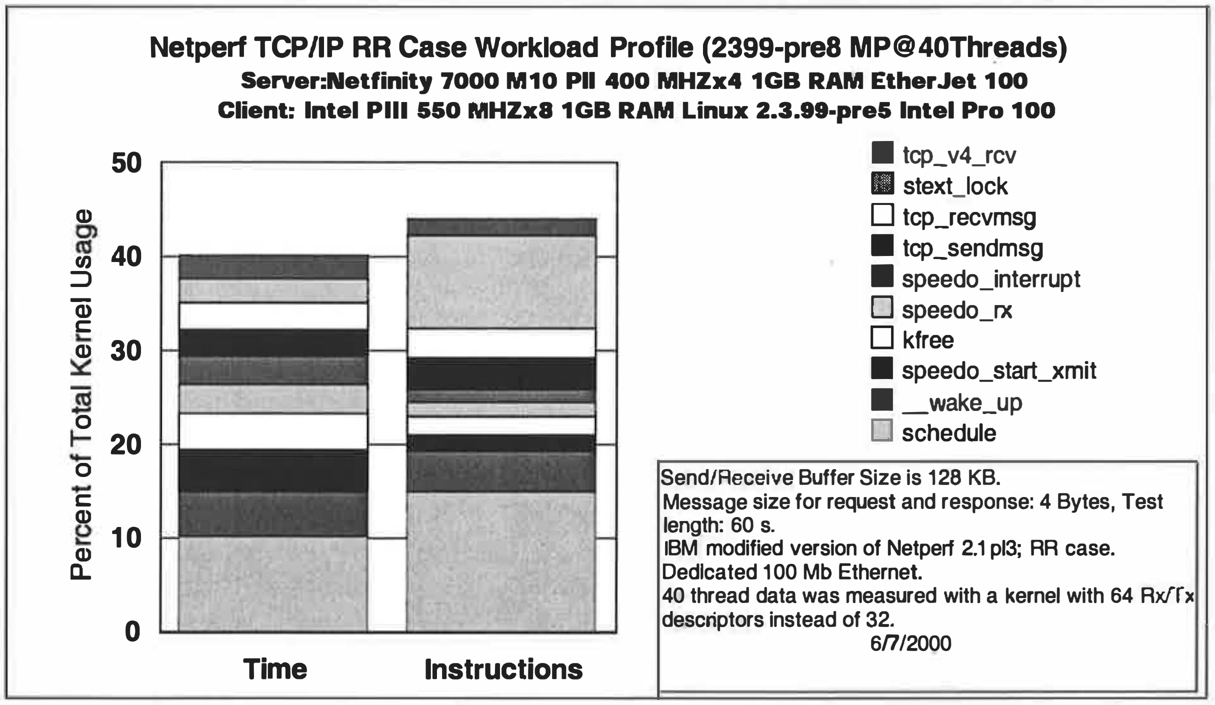


Figure 5

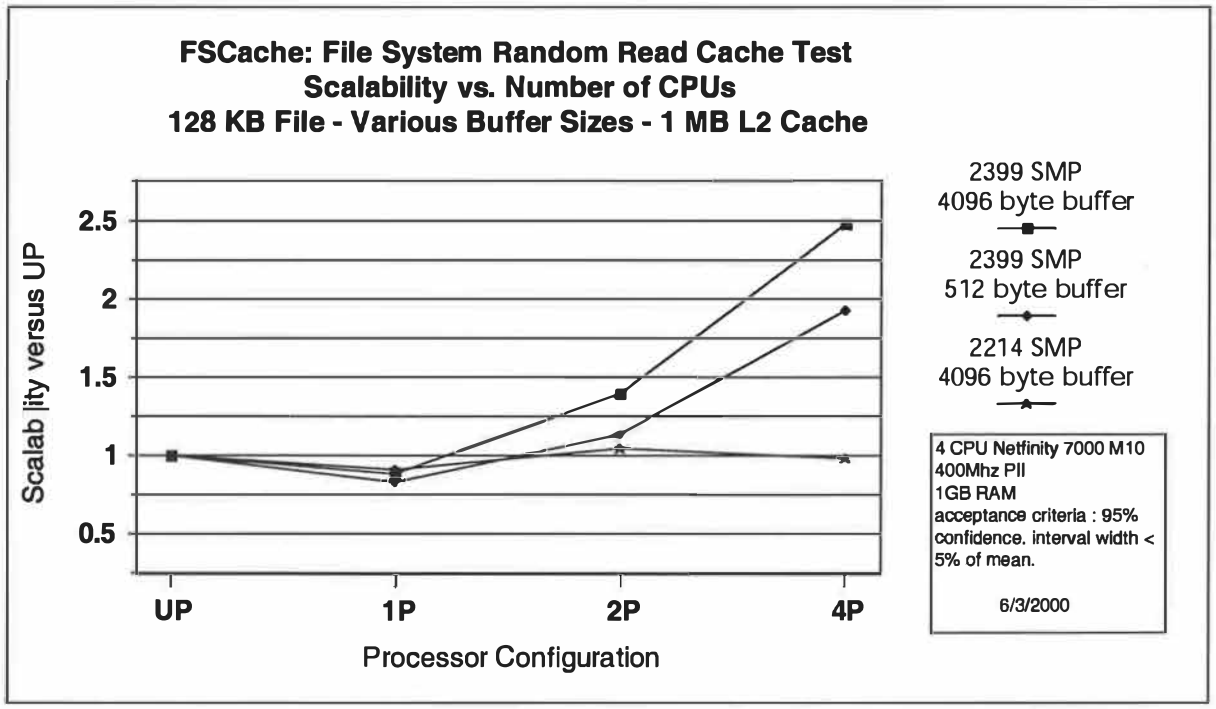


Figure 6

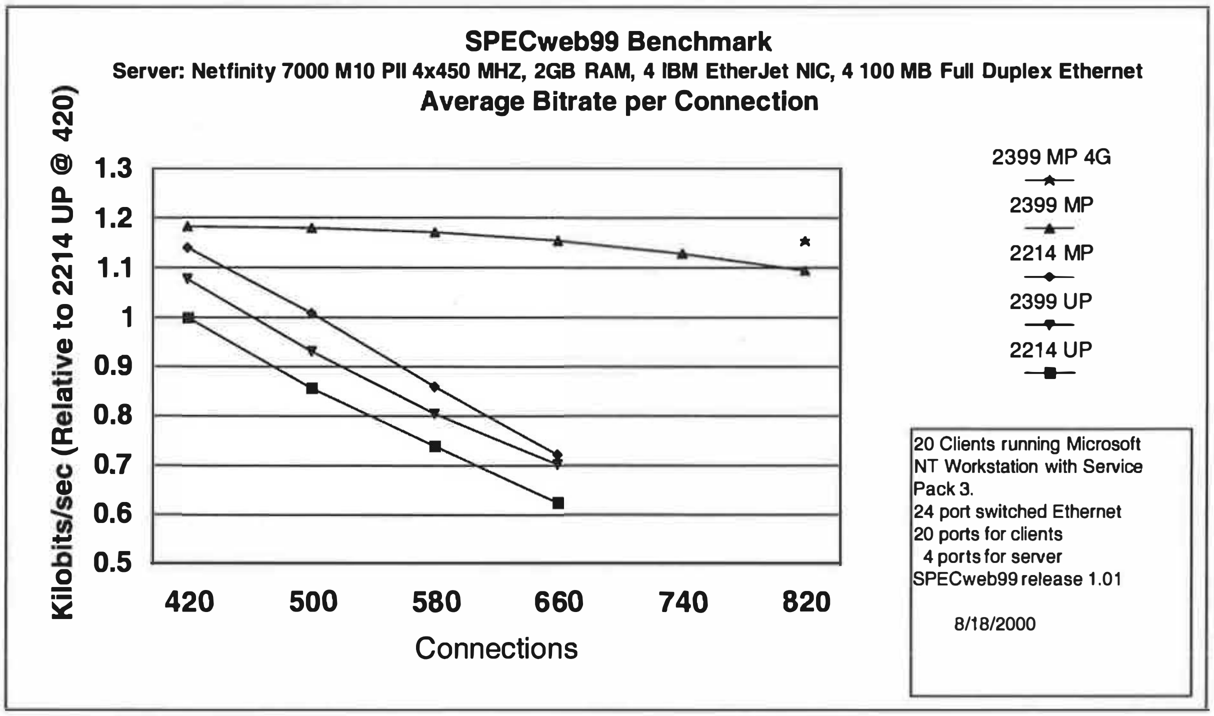


Figure 7

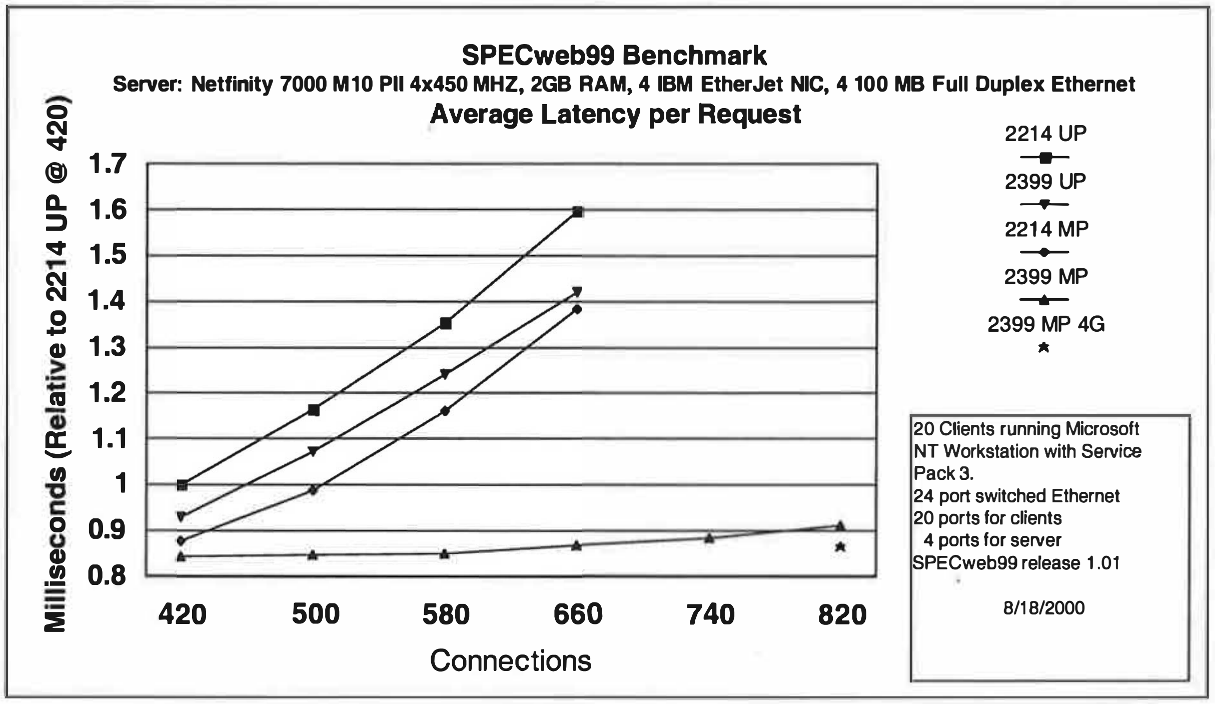


Figure 8

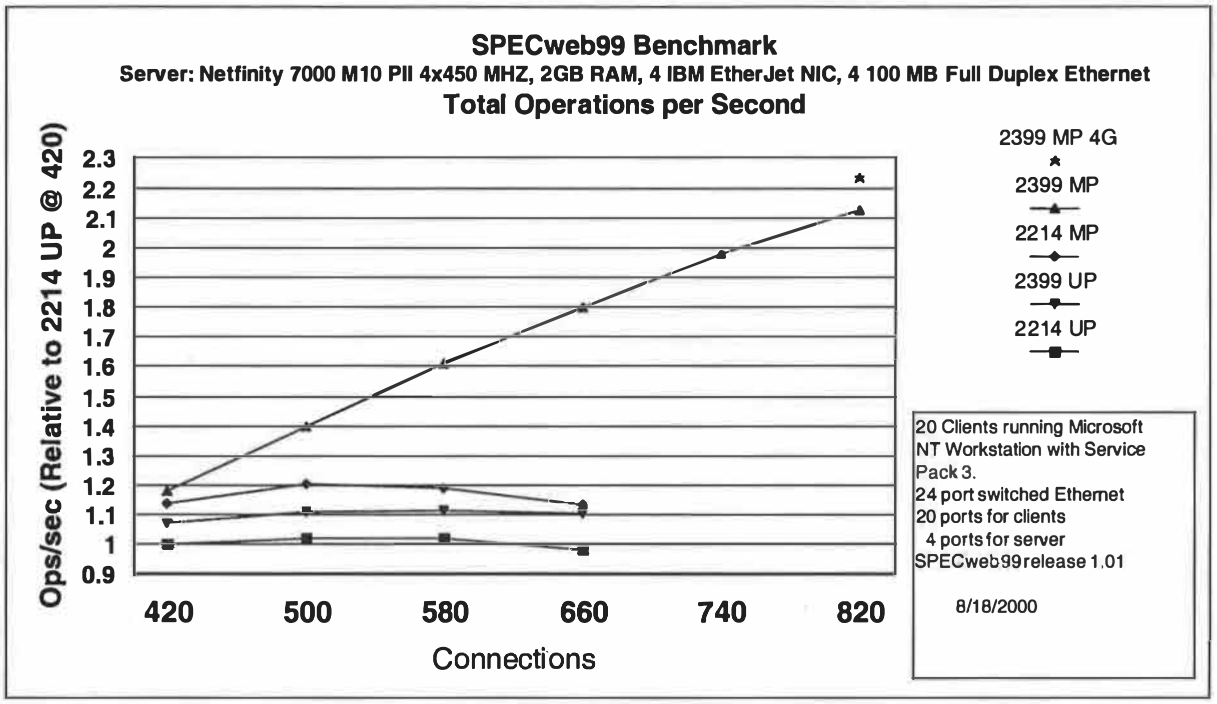


Figure 9

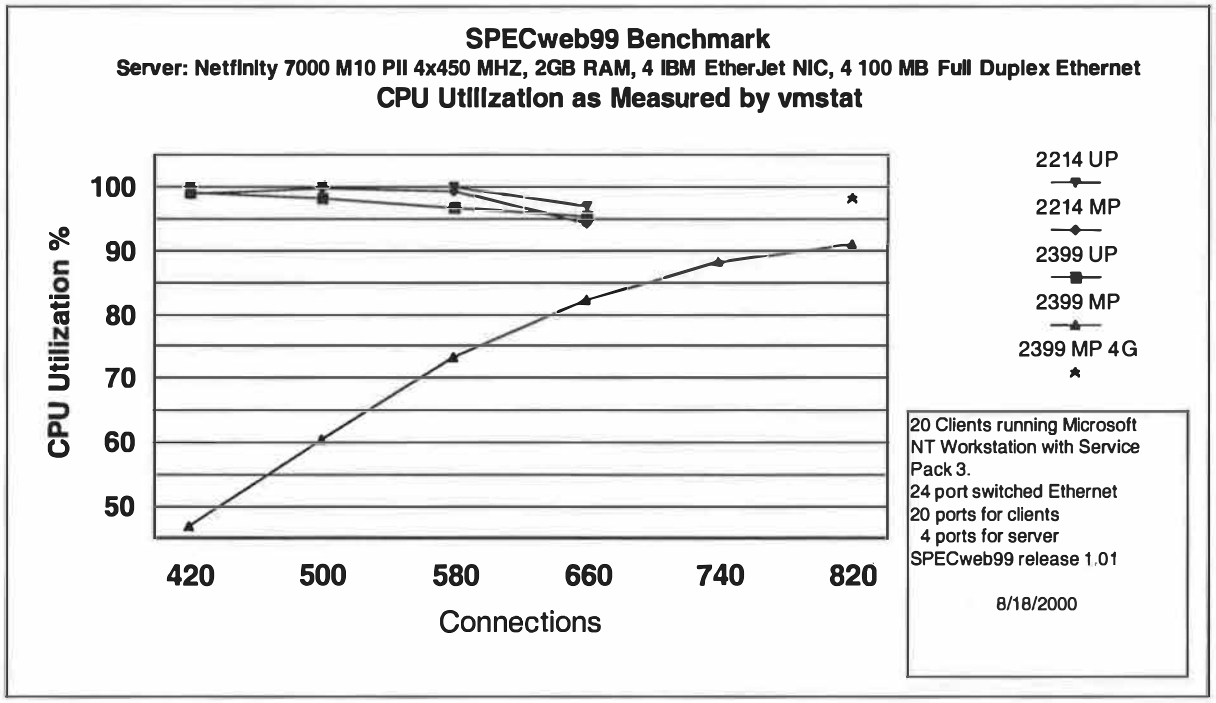


Figure 10

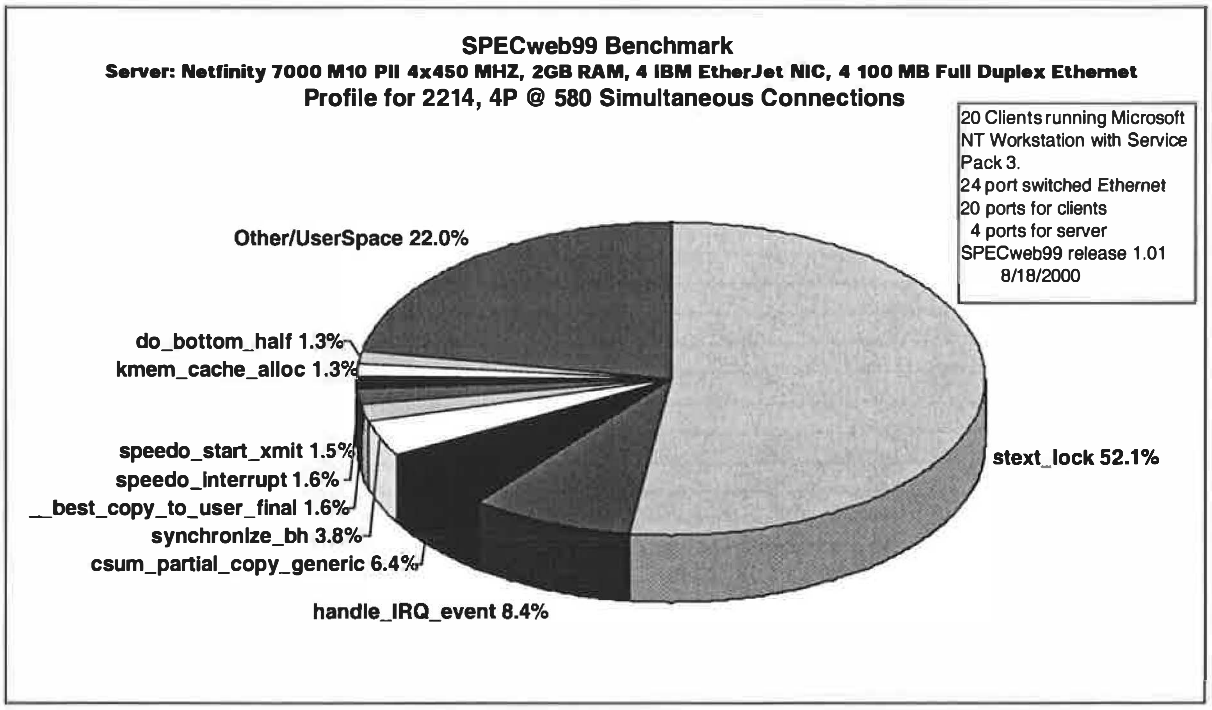


Figure 11

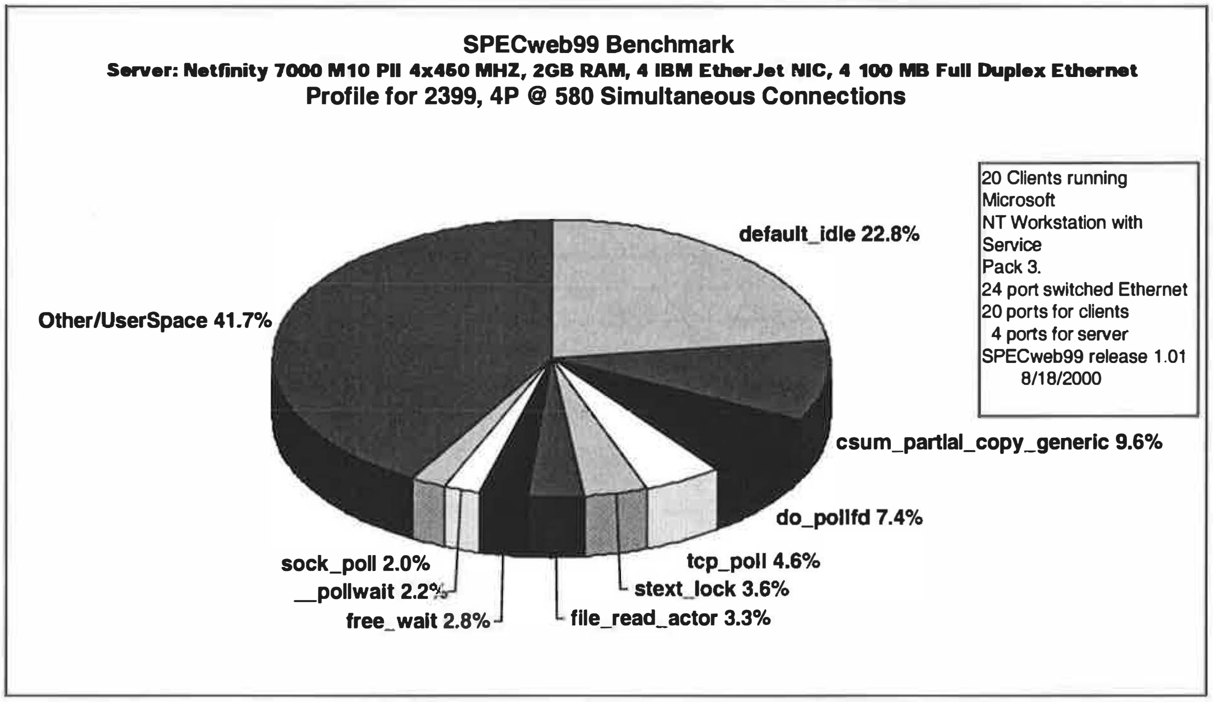


Figure 12

Design of a Very-large Linux Cluster for Providing Reliable and Scalable Speech-to-email Service

Randy Brumbaugh Todd Vernon
{rbrumbaugh | tvernon}@evoke.com

Evoke Communications
1157 Century Dr.
Louisville, CO 80027
www.evoke.com

Abstract

This paper describes experience with the design, implementation and operation of a large, scalable Linux processing cluster. One of Evoke Communications' most popular applications is a speech-to-email system, called "Talking Email." As the popularity of this service grew, meeting the demand required the design of a unique processing architecture; this paper describes the design of that architecture. The resulting architecture utilizes both pipelined and multiple levels of parallel processing to meet the requirements of processing speed, reliability and scalability. At the core of the system is a very large (200 processor) cluster of Linux systems, integrated with a very large, ultra-reliable disk storage array. The authors believe this to be one of the largest Linux cluster processing systems in existence, and also to be unique in being integrated with a very large, ultra-reliable storage system. Also discussed are operational experience and potential enhancements and changes to the architecture.

1. Introduction

Description of the Application

Among Evoke Communications' service offerings in the area of Internet Communications is a speech-to-email application. This service has proven to be both fun and popular, and consequently has required an evolution in processing architecture to handle the growth in usage.

There are two use cases for this service. In the first, a person uses a World Wide Web page to specify a recipient and is given a unique identification number and a telephone number to dial. The user dials the telephone number, which terminates in the Evoke Communications data center. There a call response system generates voice prompts, receives the user's identification number, and records the user's message digitally. The message is then compressed, encoded and stored, and the recipient is emailed a notification message including a URL that will play back the message.

In the second use case, the message recipient uses a web browser to access the URL sent via email, then receives the message as an audio stream. This may be repeated multiple times; the URL will remain valid for 30 days.

The application is designed as an engine that may be integrated to enhance other services. An example of this is Blue Mountain Arts (http://www.bluemountain.com). They use the Evoke engine to allow senders to add speech messages to some of their online greeting cards.

Design Criteria

Although the basic data flow for a single message is fairly simple, there are several challenging complications. The task was to implement a speech-to-email processing engine that could meet the following design goals:

The system must handle a large number of simultaneous incoming calls.

Message recording is initiated asynchronously by users through the Public Switched Telephone Network (PSTN). Although the front-end processing does not require huge computational resources, specialized telephony hardware is required to demultiplex voice channels from the carrier, to detect dual-tone multi-frequency (DTMF) digits, and to communicate with PSTN signaling. In addition, the hardware is dedicated to real-time service for the duration of the call; each call requires a continuously associated processor while recording speech.

The processing and storage architecture must scale to add additional capacity without redesign as the system load approaches current capacity.

This scaling must include several dimensions: number of simultaneous calls, total call throughput, and storage capacity.

The system must elegantly incorporate third-party processing packages.

Particularly in the audio encoding/transcoding and compression steps, the ability to use applications developed by others is key to success.

The system must be reliable and maintainable.

It is essential to ensure that messages are available for retrieval for an adequate length of time and the risk of message loss due to equipment failures is minimized. This is designed to be production system, available 24 hours / 7 days, with significant investment in supporting personnel, monitoring systems, and backup power. The design must permit parts of the system to fail or be taken out of production for maintenance, without losing the capability to accept incoming messages (although capacity may be reduced temporarily).

Why Linux?

Several factors led us to choose Linux as the operating system at the core of the processing cluster. First, experience with Linux has shown it to be reliable and maintainable, key attributes required. Our in-house experience with Linux also provided considerable on-site resources and expertise with the operating system and configuration. Linux also is a general purpose computing resource supporting a wide variety of packages for audio processing and compression. Not only does this allow for the current design, but provides for upgrades in the future.

But perhaps most importantly was the scalability and relatively inexpensive cost of increasing the number of processing nodes. Because many aspects of the system were difficult to simulate or predict exactly, this allowed us to adopt a philosophy of over-provisioning processing resources to meet unforeseen loads, rather than performing detailed simulation and modeling to try to optimize each segment of the processing pipeline. As a result, the central transcoding cluster intentionally contains more processing power than needed at present, but choosing Linux allowed us to make this choice to minimize the impact of complications or future changes that require more processing than projected.

2. Basic Architecture

This section describes the architecture for handling a single call, and the decomposition of that architecture into a pipeline, as well as descriptions of each processing step the data encounters in flowing through the pipe.

Process

From this point on we will focus on the architecture of the speech-to-email engine, neglecting the web interactions to initiate and play the message, as well as several significant database transactions. The web servers on both sides of the message are actually small Linux processor clusters as well.

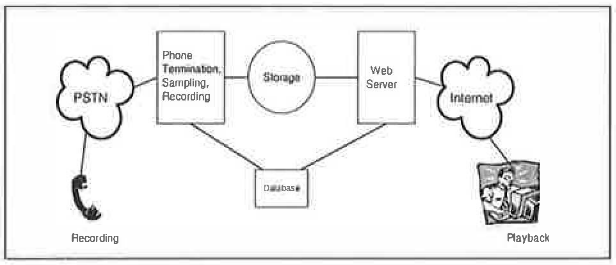


Figure 1: Simplified Message Flow Pipeline

The process required to record a speech message from a user and deliver it is shown graphically in Figure 1. The basic steps are:

1. A user's call is answered.
2. The user keys a series of pre-assigned DTMF digits to match the message to appropriate database records.
3. The message is digitally recorded, compressed and stored.
4. Email is sent to the recipient, along with a URL to retrieve the stored voice message.

Pipeline Decomposition

In designing an architecture to accomplish the task described above, the process was initially broken down into several segmented steps in a pipeline. The model used is "engines within engines," wherein the entire speech-to-email processing engine is segmented into sub-tasks, and engines designed to accomplish each of these.

In breaking up the processing task, two primary aspects were considered: function and time. The functional analysis showed that handling incoming telephone calls required specialized telephony hardware, while encoding audio into a compressed format suitable for streaming on the Internet is functionally quite different.

Similarly, timing requirements were examined, both to satisfy individual users, and to meet overall system throughput requirements. For example, user voice messages must be recorded in real-time as the user speaks; however, immediately encoding the speech into the final compressed format in real-time would require huge computational resources dedicated to each call.

The solution was to use two separate engines with an intermediate short-term storage step in-between. This meets both sets of timing and function requirements and has several advantages. In the initial step, the message is sampled and stored digitally by a dedicated telephony processor. The recorded messages are stored as uncompressed digital audio files on a shared scratchpad disk array.

The next step performs compression and transcoding into a format suited for long-term storage and streaming playback over the Internet. This step uses the cluster of Linux systems to perform transcoding faster then real-time. This asynchronous, faster than real-time processing allows the number of Linux nodes to be significantly less than the number of incoming voice channels.

Next, the compressed audio file is stored on a large, ultra-reliable storage unit. Finally, several streaming audio servers connected to the Internet provide message retrieval and streaming playback to recipients when they select the URL for playback.

3. Scaling, Clustering and Parallel Processing

This section describes the architectural modifications needed to make the process both capable of handling multiple simultaneous calls, as well as making it scalable as call volume increases. The system takes advantage of the fact that it is an engine dedicated to a single task, and pipeline processors can be dedicated to single tasks as well. Parallel processing of the SPMD (Single Program Multiple Data) type is used both within the pipeline, and to add multiple parallel pipeline structures.

Parallel processing technology can be applied to a system to increase either the processing power, the reliability or both. In this application the emphasis was placed on increasing processing power, although many reliability benefits are realized as well; most significant being the reduction of single-point nodes whose failure would halt all processing.

Parallelism within pipeline

To address the twin challenges of handling simultaneous calls and adequate processing throughput in the engine, the functions of the pipeline at each stage are handled by multiple processors working in parallel.

The first step taken was to parallelize the segment of the pipeline handling incoming calls. The selection of number of processors in this stage of the pipe largely determines the number of simultaneous calls the pipeline is able to handle, although it is important that subsequent processing stages are designed to handle the volume of work being ingested. This stage of the process does not require great computational power; it does require specialized functions and a continuous connection to the user and real-time capture of the voice message. To accomplish this, a large number of small proprietary telephony processors are assigned the task of answering and recording the incoming calls.

The heart of the system is the next stage of the pipeline: the transcoding engine. This takes the raw digital audio files and performs processing and compression so that they may be stored efficiently and replayed through a streaming Internet server. This is where the cluster of Linux processors is used. The number of parallel processors needed for this stage is primarily determined by the workload generation capability of the telephony processors, but overall throughput and insuring excess capacity are also factors. The current transcoding application is able to convert a message in approximately one-tenth of the time taken to record it, but other transcoding and processing may be added in the future without requiring additional transcoding processors.

Storage, communication and synchronization between pipeline processing segments are accomplished via files shared on a network file system (NFS) served over high-speed Ethernet links.

The resulting architecture is illustrated in Figure 2.

Parallel Pipelines

Once the basic pipeline architecture is established, additional capability is added by cloning the original pipeline into additional pipelines, each using the same architecture as the first, in parallel. These pipelines operate asynchronously and independently.

Having multiple parallel pipelines increases the reliability and maintainability of the overall speech-to-email engine. Any single pipeline or part of a pipeline may fail or be taken offline for maintenance, but the engine still operates, although at somewhat reduced capacity.

This is useful for performing upgrades or changes to the engine in a phased manner, without interrupting service. For an upgrade, a single pipeline is removed from service and the changes to hardware or software are made in that pipeline only. Incoming calls can be selectively routed to the upgraded pipeline to evaluate the new configuration. This is typically done in phases, starting with internal calls and building to production calls as confidence increases. Once the new configuration is validated, the process can be repeated for the remaining pipelines. If the new configuration proves to be faulty, it is similarly easy to remove that processing pipe from service again and restore it to the original condition.

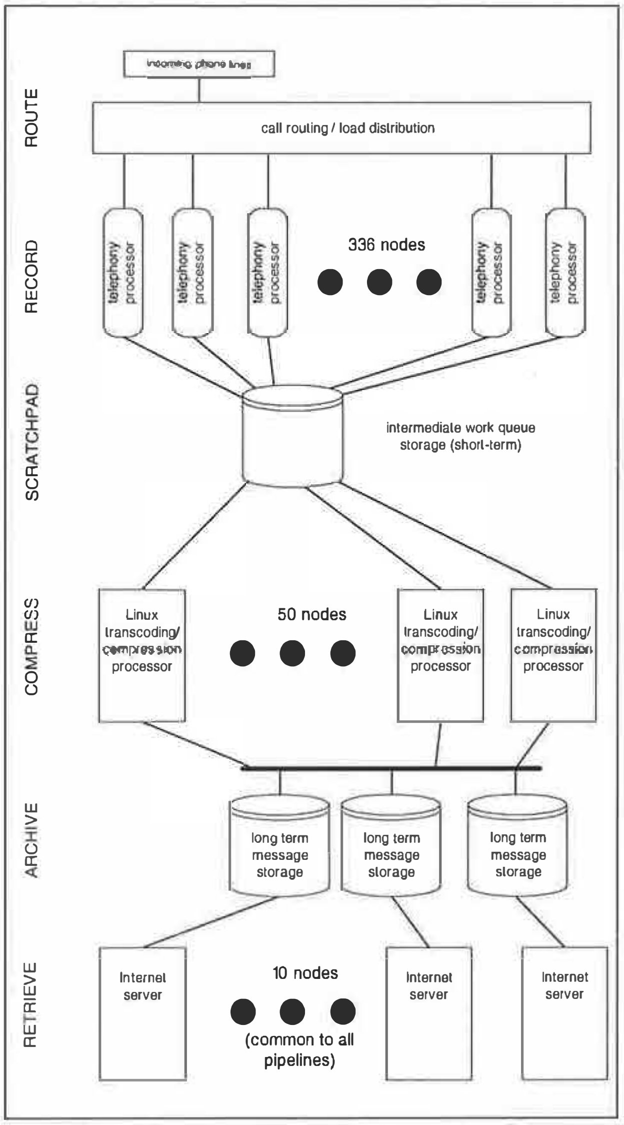

incoming phone lines
ROUTE
call routing / load distribution
RECORD
telephony processor
336 nodes
SCRATCHPAD
intermediate work queue storage (short-term)
COMPRESS
Linux transcoding/ compression processor
50 nodes
ARCHIVE
long term message storage
RETRIEVE
Internet server
10 nodes
(common to all pipelines)

Figure 2: Architecture of Single Pipeline

Load Balancing and Process Distribution

Load balancing and process allocation is required at several steps in this processing cluster. When calls initially arrive, they are assigned to specific processors for the initial setup and voice recording. A set of routing rules was designed when the incoming telephone lines were provisioned; these rules implement a pseudo-random assignment of calls to processors, and thus to pipelines. Once a call is routed to a voice recorder, it is processed in the pipeline downstream of that recorder.

The next step requiring balancing is after the message is recorded, and one of the Linux compression/transcoding machines must be assigned to retrieve and process the audio file. As each incoming message is recorded to digital scratchpad storage, it is assigned to a queue associated with a specific processor in the transcoding cluster. This assignment is determined by a policy-based scheme designed both to balance workload among the cluster processors, and to distribute the load across processors. The effect appears as a random assignment; consecutive incoming messages from to a voice processor are not likely to route to the same transcoding processor.

Storage

A key element of the system is the storage of the compressed audio files for streaming playback on demand, when a recipient selects the URL sent to them in email. The storage system must be capable of holding a large amount of data, be reliable and upgradeable. The system chosen is a commercial network file server that appears as several shared disks, mapping to each of the transcoding boxes.

A part of the storage array is also used as the work queue storage between the recording and compression steps. This array is mapped to the associated processors as a collection of NFS mount points.

4. Analysis of Design and Performance

Since being put into production in January 2000, the system has proven to be reliable and stable, with few unscheduled downtimes and very few lost messages. It has met design goals, and user feedback has been positive. But nothing is as convincing as personal experience, so the reader is invited to try it: go to the Evoke Communications Talking Email web site (http://www.evoke.com/Poweredby/TalkingEmail/), click on "Send a Talking Email," and send a message to a friend.

Capability and Performance

The theoretical capability boundaries of the speech-to-email engine are: over 1300 simultaneous messages, a throughput in excess of 1,000,000 messages per day, each message retained for 30 days (see figure 3). This capacity is designed to meet current and foreseeable future needs, including extra capacity to meet demand even when the system is degraded due to failures or maintenance downtime.

Testing these theoretical capability boundaries presents some challenges. The system is operating in production in an environment requiring continuous availability. Planned outages or maintenance require planning and coordination. Since the purpose of the system is production service and high availability rather than research, taking any part of the system out of service for research is not a popular idea. And certainly removing the entire cluster from service for evaluation is not an option.

Parts of the pipeline were evaluated during design and prototyping phases of the project. For example, the telephony processors' ability to handle incoming call loads was confirmed. It has also been measured that the average incoming message is 30 seconds in length, and is average transcoding time is 3 seconds. Transcoding latencies have not been characterized, but have been observed to be on the order of milliseconds and not dependant on incoming call loading at the call volumes experienced in the life of the system.

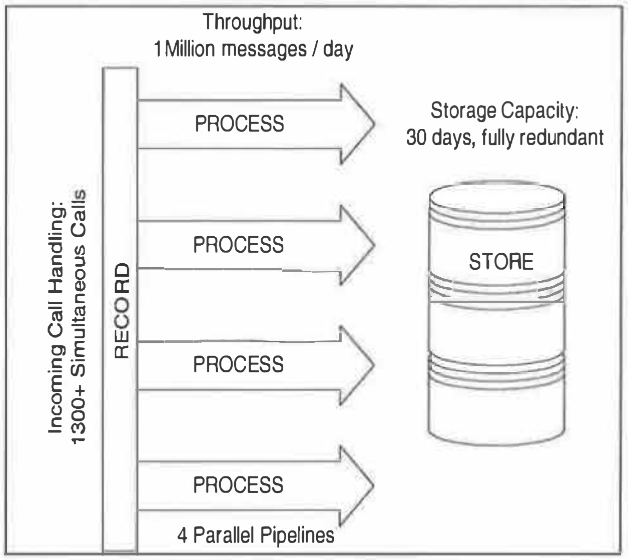


Figure 3: Theoretical Performance Bounds

The system was put in to production with a load far below the maximum of which it is theoretically capable, and it is designed to accommodate steady growth in usage for several years. As a consequence, although the usage has been steadily increasing, it has never approached the capacity of the engine. Load testing is difficult, because the system capability is so large that generating a significant load would require a similarly large test system. Further, since the system is in production, any testing that might interfere with production usage or risk a crash is considered unwise.

The system is monitored continuously for usage, and each node of the cluster runs a monitor process that reports to a workstation for detection of failed nodes and other problems. The associated databases also maintain log files that can be mined for trends and indicators of cluster and processor health.

It should be noted that this engine has been in continuous use, in a production environment, since January 2000. During that time it was available around the clock, largely operating independently and unattended. The proven capability to meet reliability and autonomous operation requirements is a critical indicator of the performance of the architecture and implementation.

Weaknesses

The primary weakness that has been identified is the method of assigning work to the transcoding cluster processors. Although no significant failures have occurred, it is recognized that the system has a failure mode that could result in delayed (but not lost) messages. The difficulty is that each message is assigned to a transcoding processor queue when it is initially recorded from the telephone. Should any transcoding processor fail, the work assigned to it will not be performed until the machine is repaired or an operator intervenes. No work is lost, but messages may be delayed in queue. Several options for improving this are being considered. These include shared transcoding queues, centralized work queue monitoring, and opportunistic transcoding processors in a workpile setup, where processors actively search for raw sound files needing conversion. These all have advantages and disadvantages when compared with the current set of static work queues.

5. Future Directions

Although the system has proven to be reliable and stable, we continue to look at technologies and techniques to improve it, or to use in future generations.

Clustering Technologies and Architectures

We have evaluated various architectures used in other clustering systems, as well as PVM and MPI communication technologies. In particular, some aspects of Beowulf clusters were intriguing because of their popularity. It was decided that these clustering technologies were more suited to general purpose computing workstations, where our application is a massive application of a single-purpose processing engine. However, we continue to watch developments and reports in the field of Linux clustering.

Changing Audio Encoders

One of the theoretical advantages of using a Linux-based cluster is now being realized. New audio codecs have become available which offer improvements in many aspects of storage and content delivery. Support for these codes in Linux is readily available, and the fact that the transcoding array, although running a single task, is based on general-purpose computers and operating system, makes changing transcoder applications and parameters relatively easy. In addition, the over-provisioning of the transcoding cluster processing resources means that there is little concern that a transcoder that requires increased processor time will reduce the throughput of the system.

Voice over IP

Use of packetized voice carried over the Internet or dedicated IP networks is steadily increasing. In the future it is likely that these networks will carry significant amounts of telephone voice traffic as well as connect directly to computers equipped with microphones and speakers. We are keenly interested in this technology and the possibility for adding this capability to our speech-to-email engine.

Shrinking

One intriguing area of technology convergence centers on the compact PCI standard. Compact PCI enclosures are now available, designed for to be ultra-reliable, and with features for supporting telephone resource cards. At the same time, an increasing number of general-purpose processors, telephony interfaces and DSP resource cards designed for telephony, are becoming available in compact PCI form-factor with Linux support. This investigation may allow us to replace the proprietary telephony processing front end with a Linux-based solution. It may also allow the call recording and compression steps to be performed in a single enclosure, reducing space, complexity and power requirements. There would still be a degree of parallel processing, since the bulk of the encoding would be done by firmware controlling digital signal processor resources.

6. References

[1] Gregory R. Andrews, "Foundations of Multithreaded, Parallel, and Distributed Programming," Reading, Massachusetts: Addison-Wesley, 2000.

[2] Rajkumar Buyya (ed.), "High Performance Cluster Computing," Vol. 1, Upper Saddle River, New Jersey: Prentice-Hall PTR, 1999.

[3] David E. Culler, Jaswinder Pal Singh, Anoop Gupta, "Parallel Computer Architecture," San Francisco: Morgan Kaufman, 1999.

[4] John P. Hayes, "Computer Architecture and Organization," New York: Mc Graw-Hill, 1978.

[5] Thomas L. Sterling, John Salmon, Donald J. Becker, Daniel F. Savarese, "How to Build a Beowulf: A Guide to the Implementation and Application of PC Clusters ," Boston: MIT Press, 1999.

Cluster	**Nodes**	**Node configuration**		
		OS	**CPU**	**memory**
Telephony Processing	1344	Specialized telephony	DSP	--------
Transcoding/ Compression	200	Linux	P3-600	500 MB
Web Server	20	Linux	P3-600	500 MB

Figure 4: Cluster Configuration Summary

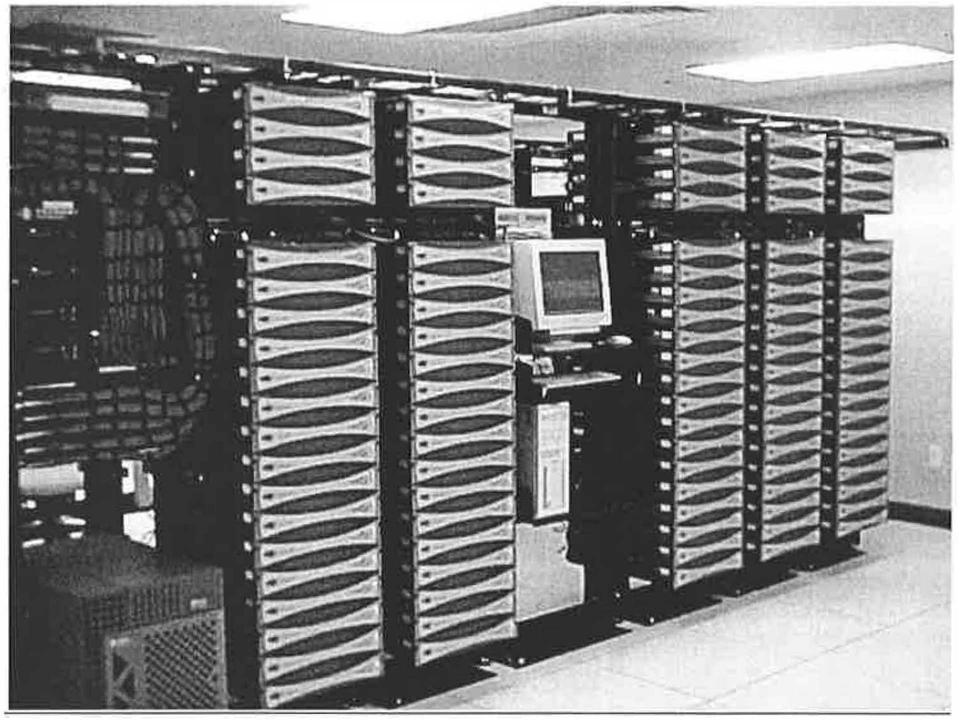


Figure 5: 200 Processor Linux Cluster for Compression / Transcoding

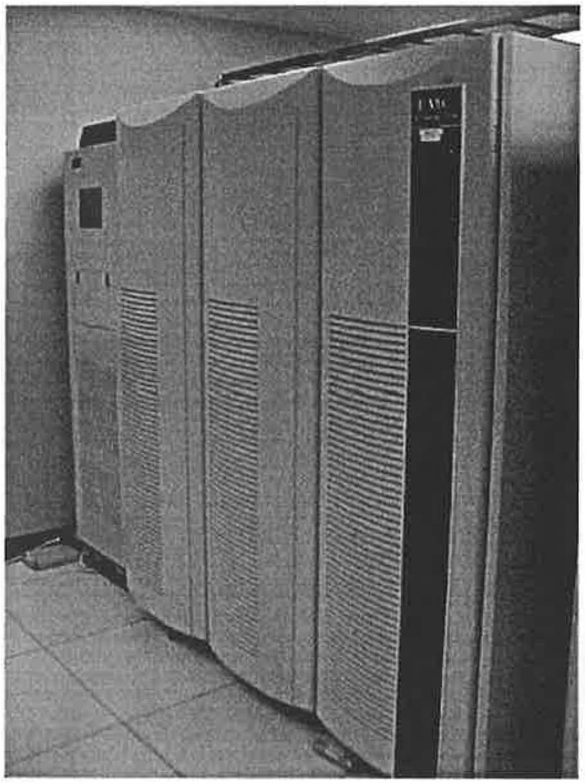


Figure 6: Message and Workqueue Storage

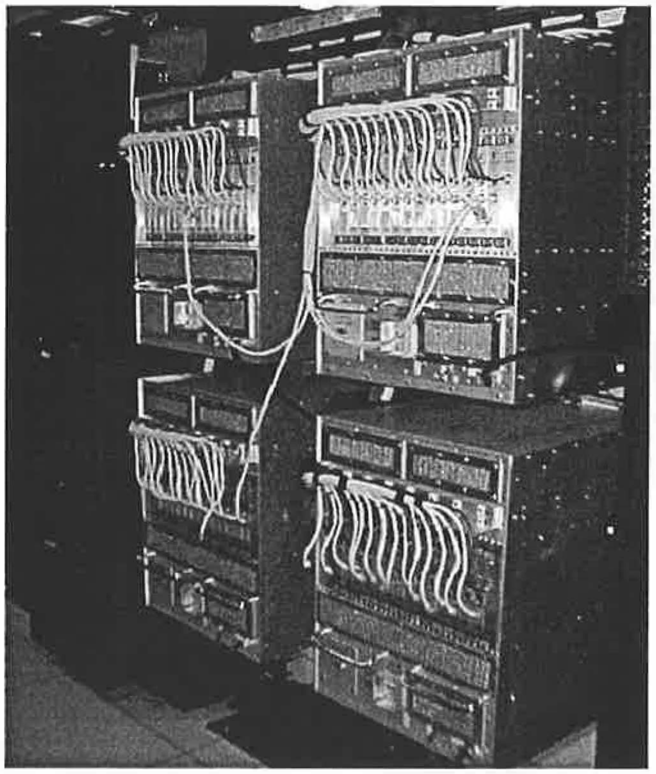


Figure 7: Telephony Processing Cluster

The Portable Batch Scheduler and the Maui Scheduler on Linux Clusters*

Brett Bode, David M. Halstead, Ricky Kendall, and Zhou Lei
Scalable Computing Laboratory, Ames Laboratory, DOE
Wilhelm Hall, Ames, IA 50011, USA, help@scl.ameslab.gov
David Jackson, Maui High Performance Computing Center

Abstract

The motivation for a stable, efficient, backfill scheduler that runs in a consistent manner on multiple hardware platforms and operating systems is outlined and justified in this work. The combination of the Maui Scheduler and the Portable Batch System (PBS), are evaluated on several cluster solutions of various size, performance and communications profiles. The total job throughput is simulated in this work, with particular attention given to maximizing resource utilization and to the execution of large parallel jobs.

1 Introduction

With the ever increasing size of cluster computers, and the associated demand for a production quality shared resource management system, the need for a policy based, parallel aware, batch scheduler is beyond dispute. To this end the combination of a stable, portable resource management system, coupled to a flexible, extensible, scheduling policy engine will be presented and evaluated in this work. Features, such as extensive advanced reservation, dynamic prioritization, aggressive backfill, consumable resource tracking and multiple fairness policies, will be defined and illustrated on commodity component cluster systems. The increase in machine utilization and operational flexibility will be demonstrated for a non-trivial set of resource requests over a range of duration, and processor count tasks.

We will use the term *large* to describe jobs that require a substantial portion (>50%) of the available CPU resources of a parallel machine. The duration of a job, is considered to be independent of its resource request, and for the purposes of this paper the term *long* will be used to identify jobs with an extended runtime of multiple hours. The opposite terms of small and short will be used for the converse categories of jobs respectively.

2. Scheduling Terminology

To clarify the nomenclature used in the descriptive sections of this work we will now include a glossary of terms, together with a brief explanation of their meaning.

Utilization and turnaround

The ultimate aim of any dynamic resource administration hierarchy is to maximize utilization and job throughput and minimize turnaround time. The aim of improving utilization can be achieved by allocating tasks to idle processors, but the task of maximizing throughput is much more nefarious, involving complex fair access decisions based on machine stakeholder rights.

Prioritization and fairness

It is the goal of a schedule administrator to balance resource consumption amongst competing parties and implement policies that address a large number of political concerns. One method of ensuring appropriate machine allocation is with system partitioning. This approach, however, leads to fragmentation of the system, and a concomitant fall in utilization efficiency. To be preferred is a method by which the true bureaucratic availability requirements can be met, without negatively impacting the utilization efficiency of the resource.

Fair share

The tracking of historical resource utilization for each user results in the ability to modify job priority, ensuring a balance between appropriate access, and maximizing machine utilization. Users can be given usage targets, floors and ceilings which can be configured to reflect the budgeted allocation request.

*This work is supported by the Applied Mathematical Sciences Program of the Ames Laboratory- U.S. Department of Energy (USDOE) under contract number W-7405-ENG-82 which is in part under the auspices of Computational Science Program of the Mathematical, Information and Computational Science Division of the Office in Advanced Scientific Computing Research at USDOE

Reservation

The concept of resource reservation is essential in constructing a flexible, policy based batch scheduling environment. This is usually a data structure that defines not only the computational node count, but also the execution timeframe and the associated resources required by the job at the time of its execution.

Resource manager

The resource manager coordinates the actions of all other components in the batch system by maintaining a database of all resources, submitted requests and running jobs. It is essential that the user is able to request an appropriate computational node for a given task. Conversely, in a heterogeneous environment, it is important that the resource manager conserve its most powerful resources until last, unless they are specifically requested. It also needs to provide some level of redundancy to deal with computational node failure and must scale to hundreds of jobs on thousands of nodes, and should support hooks for the aggregation of multiple machines at different sites.

Job scheduler/ Policy manager

The job scheduler takes the node and job information from the resource manager and produces a list sorted by the job priority telling the resource manager when and where to run each job. It is the task of the policy manager to have the flexibility to arbitrate a potentially complex set of parameters required to define a fare share environment, yet retain the simplicity of expression that will allow the system administrators to implement politically driven resource allocations.

Job execution daemon

Present on each node, the execution daemon is responsible for setting up the node, servicing the initiation request from the resource manager, reporting its progress, and cleaning up after the job termination, either upon completion or when the job is aborted. It is important that this be a lightweight and portable daemon, allowing for rapid access to system and job status information and exhibit a low overhead of task initiation, to facilitate scalable startup on massively parallel systems.

Co-allocation

A requirement for an integrated computational service environment is that mobile resources, such as software licenses and remote data access authorizations, may be reserved and accessed with appropriate privileges at execution time. These arbitrary resources need to be made transparently available to the user, and be managed centrally with an integrated resource request notification system.

Meta-scheduling

A meta-scheduler is a technique of abstraction whereby complex co-allocation requests and advanced reservation capabilities can be defined and queried for availability before the controlling job begins execution. This concept ties in naturally to the requirement for data pre-staging since this can be considered as merely another resource.

Pre-staging

The problems of coordinating remote data pre-staging or access to hierarchical storage can be obviated by an intelligent advance reservation system. This requires integration with the meta-scheduler and co-allocation systems to ensure that the initiation of the setup phase is appropriately timed to synchronize with the requested job execution.

Backfill scheduling

The approach of backfill job allocation is a key component of the Maui scheduler. It allows for the periodic analysis of the running queue and execution of lower priority jobs if it is determined that their running will not delay jobs higher in the queue. This benefits short, small jobs the most, since they are able to pack into reserved, yet idle, nodes that are waiting for all of the requested resources to become available.

Shortpool policy

The shortpool policy is a method for reserving a block of machines for expedited execution and turnaround. This is usually implemented during workday hours and can predictively assign currently busy nodes to the shortpool if their task will finish within the required window (usually under two hours).

Allocation bank

The concept of an allocation bank is essential in a multi-institution shared resource environment [1]. It allows for a budgeted approach to resource access, based on a centralized user account database. The user account is debited upon the execution of each job, with the individual being unable to run jobs once the account has been exhausted.

Reservation security

An essential area of research centers on authentication and security of remotely shared resources. Issues such as secure interactive access and user authentication have been addressed and resolved to a large extent, but the issue of delayed authentication and inter-site trust are still subjects for research. The important factor of non-repudiation needs to be addressed in order to validate the source of a submission.

Job expansion factor

This is a way of giving small time limit jobs a priority over larger jobs by calculating their priority from the sum of the current queue wait time and the requested wall time relative to the requested wall time.

Job Efficiency

This is the percent of the requested time actually used by the job. For simple schedulers this factor has little effect on the overall scheduling performance. However, for the backfill portion of the Maui Scheduler this factor has a much more significant influence, since low job efficiencies cause inaccurate predictions of holes in the reservation schedule. The best ways to improve this factor are user education and resource monitoring.

Quality of service

The Maui Scheduler allows administrators fine grain control over QOS levels on a per user, group and account basis. Associated with each QOS is a starting priority, a target expansion factor, a billing rate and a list of special features and policy exemptions. This can be used impose graduated access to a limited resource, while ensuring maximum utilization of idle computational assets.

Downtime scheduling

One important advantage of a time based reservation system is that scheduling down-time for repair or upgrade of running components is easily performed. This is convenient for the current clusters, but will become essential as the size and production nature of parallel clusters continues to increase.

SMP aware queuing

The debate over the utility of multiple processors per computational node continues to rage. It is clear, however, that the incremental cost of additional CPUs in a node is less than a concomitant increase in the total number of nodes. The question is how to exploit the co-location advantage of data between SMP CPUs, and to expose this potential performance enhancement to the user in a consistent manner via the scheduling interface. There are several different approaches available to exploit SMP communications (Pthreads, Shmem etc.), but this topic is beyond the purview of this work.

3. Testbed Hardware

3.1 Linux 64 node

The largest test environment considered in this work is a cluster of 64 Pentium Pro machines with 256 MBytes of RAM, connected by a flat 44 Gbit/sec Fast Ethernet switch. The cluster was constructed in accordance to the Scalable Cluster Model [2] with the file server and external gateway node utilizing a dedicated Gigabit Ethernet connection to the switch for improved performance. The compute nodes are running a patched 2.2.13 Linux kernel with the server nodes providing both Fortran and C compilers with MPI and PVM message passing libraries. Version 2.2p11 of the PBS server runs from the file server node, along with the Maui scheduler version 2.3.2.12.

3.2 Compaq 25 node

The other cluster from which results will be reported consists of 25 Compaq 667 MHz Alpha XP1000 machines, 15 of which have 1024 MB of RAM and 10 have 640 MB of RAM. One of the 25 nodes has reduced scratch disk space and is thus limited to small jobs. These machines are connected by Fast Ethernet and run the Tru64 Unix operating system. This configuration provides a fairly challenging test for the scheduler since there are three different node resource levels available for users to request. Since we ported the Maui Scheduler PBS plugin to Tru64 Unix several months ago, the cluster has been running in production mode, executing parallel computational chemistry jobs using the GAMESS [3] code. We will be using this cluster to illustrate the pitfalls of real-world environments, and to highlight some of the measures that can be taken to ameliorate certain user shortcomings.

4. Batch scheduler description

4.1 Background

Since clusters and cluster-like systems have been around for several years, there have been multiple queuing systems tried out and several are currently in wide use. Among these are the Distributed Queuing System (DQS), Load Sharing Facility (LSF), IBM's LoadLeveler, and most recently the Portable Batch System (PBS). Each of these systems has strengths and weaknesses. While each of these works adequately on some systems, none of them were designed to run on cluster computers. Currently the system with the best cluster support is PBS. Thus we will consider PBS using its built-in scheduler compared with the addition of the plugin Maui scheduler.

4.2 PBS

Portable Batch System is a POSIX compliant batch software processing system originally developed at NASA's Ames research center for their large SMP parallel computers [4]. It has the advantage of being configurable over a wide range of high power computer architectures, from heterogeneous clusters of loosely coupled workstations, to massively parallel supercomputers. It supports both interactive and batch mode, and has a user friendly graphical user interface.

Recently the focus of development has shifted to clusters and basic parallel support has been added. In addition, the Maui scheduler has been ported to act as a plugin scheduler to the PBS system. This combination is proving successful at scheduling jobs on parallel systems. However, since PBS was not designed for a cluster-like computer, it lacks many important features. For instance, while the resource manager and scheduler are able to reserve multiple processors for a parallel job, the job startup, including the administrative scripts, is performed entirely on one node.

PBS includes several built-in schedulers, each of which can be customized for the local site requirements. The default is the FIFO scheduler that, despite its name, is not strictly a FIFO scheduler. The behavior is to maximize the CPU utilization. That is, it loops through the queued job list and starts any job for which fits in the available resources. However, this effectively prevents large jobs from ever starting since the required resources are unlikely to ever available. To allow large jobs to start, this scheduler implements a "starving jobs" mechanism. This mechanism initiates when a job has been eligible to run (i.e. first in the queue) longer than some predefined time (the default is 24 hours). Once the mechanism kicks in, the scheduler halts starting of new jobs until the "starving" job can be started. It should be noted that the scheduler will not even start jobs on nodes which do not meet the resource requirements for the "starving job".

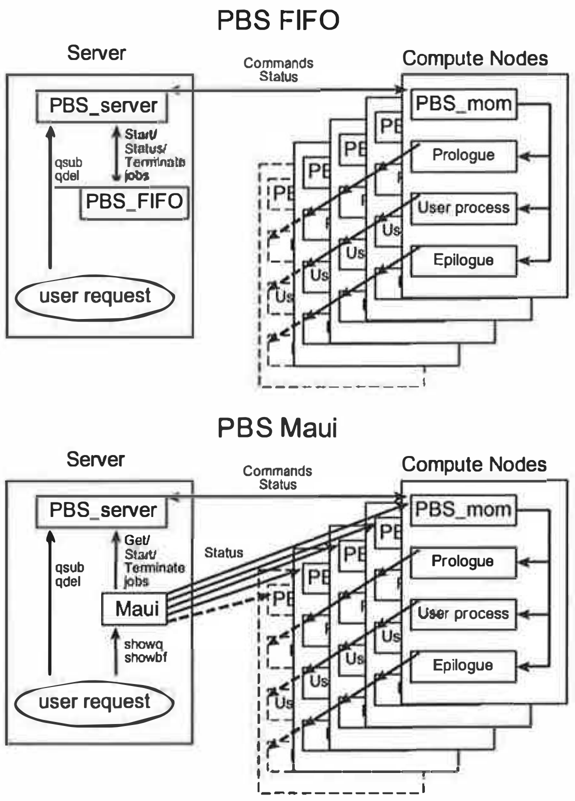


Figure 1. Schematic of the interaction profile between PBS running the FIFO scheduler and the Maui Scheduler.

4.3 Maui

Perhaps the most complete system currently available is the IBM LoadLeveler software developed originally for IBM's SP machines, but now available on several platforms (Linux is not supported). While LoadLeveler provides many useful features, its implementation leaves a lot to be desired. For instance the scheduler was immediately recognized as inadequate, since its poor parallel scheduling resulted in low total machine usage due to many idle nodes waiting for future jobs. To solve this problem the Maui Scheduler [5] was written principally by David Jackson for the Maui High Performance Computer Center. This scheduler has proven to be a dramatic success on the SP platform, so much so that it is now used in place of the default scheduler in LoadLeveler at many SP installations. LoadLeveler is probably the only currently available package, which was designed for a parallel computer from the beginning and thus addresses many of the requirements listed above. Figure 1 illustrates the difference between the PBS FIFO and PBS with the Maui scheduler in place.

The key to the Maui Scheduler is its wall-time based reservation system. This system orders the queued jobs based upon priority (which in turn is derived from several configurable parameters), starts all the high priority jobs that it can, and then makes a reservation in the future for the next high priority job. Once this is done, the backfill mechanism attempts to find lower priority jobs that will fit into time gaps in the reservation system. This gives large jobs a guaranteed start time, while providing a quick turn around for small jobs.

5. Evaluation description

5.1 The Simulated Job Mix

The right mix of jobs for any simulation is nebulous at best. Nothing is better than a real job mix from the user community in question, but that is impossible to reproduce due to user dynamics. User resource requests vary directly with their needs and cycle with external forces such as conference deadlines. To this end we have defined a job mix that fits a rough average of what we have observed on our research clusters and on the MPP systems available at supercomputer centers such as NERSC [6].

The job mix has Large, Medium, Small, Debug, and Failed jobs. Each job has a randomized set of the number of processors (nproc), the time actually spend doing work (work time), the time requested from the resource management system (submit time) and a submission delay time (delay time). Large, Medium, and Small jobs have a work time that is 70% or more of the submit time. Submit time is always greater than or equal to the work time. Large jobs are those that have nproc > 50% of those available. Medium jobs have nproc between 15% and 50% of the available nodes. Small jobs are those with nproc between 30% and 15% of available nodes. Debug jobs have a work time that is greater than 40% of the submit time but use less than 10% of the available processors. Failed jobs are defined by a work time that is less than 20% of the submit time.

Close inspection of these parameters will show that not all jobs generated by a randomized nproc, work time, and submit time, fall into these categories. One further target constraint is that Large jobs are 30% of the total set of jobs, Medium jobs are 40%, Small jobs are 20%, Debug Jobs and Failed jobs are both 5%. Jobs are randomly generated and then classified as Large, Medium, Small, Debug, Failed or "undefined" jobs. Undefined jobs are automatically rejected and others are added only if their addition will not increase their constrained classification above the limits outlined above.

In the 76 jobs of the job mix used in these simulations, 480 random jobs were generated. The resultant mix from this defined job mix algorithm yielded 28.95% Large, 40.79% Medium, 19.74% Small, and 5.26% Debug and Failed jobs. In actual numbers this corresponds to 22 Large, 31 Medium, 15 Small, 4 Debug, and 4 Failed jobs.

The randomized delay time has the effect of jobs being submitted in a random order to the batch system. The first job generated will not necessarily be the first job submitted. All of the job mix data is available online [7]. The exact same job mix was used with each scheduler setup, PBS/FIFO, PBS/MAUI and PBS/MAUI with backfill turned off.

5.2 Users and the Job Mix.

The defined job mix does not consider user interaction currently. There are no automatic or post submitted jobs that fit any gaps in the system as the scheduler runs jobs, e.g., jobs to fit the backfill mechanism available in some schedulers. Furthermore, it is quite typical for users to simply submit jobs with the maximum allowed time for the queue in question. Our simulation assumes users can predict the resources needed with reasonable accuracy. All jobs are submitted after 180 minutes from the start of the simulation. This does not match the constant influx of jobs on our research cluster or at any supercomputer center.

6. Test results

6.1 Simulation results

Perhaps the most significant result and the simplest are the total run time for each of the scheduler configurations as shown in Table 1.

Table 1.

Scheduler	Total run time (Hours)
PBS FIFO	71.1
Maui Scheduler	66.75
Maui without backfill	66.71
Theoretical Minimum	53.6
Sequential Maximum	90.2

Table 1 includes a Theoretical Minimum time which is simply the total number of node-wall hours divided by 64 (the number of available CPUs). This is clearly not an achievable value since it ignores the packing efficiency of the jobs. Conversely the Sequential Maximum represents the maximum total time the tests would take if they were simply run in a FIFO fashion with no attempt to overlap jobs.

Obviously both schedulers do substantially better than the Sequential Maximum, and the Maui scheduler does substantially better than the PBS FIFO scheduler. It may, however, be surprising that the backfill scheduler is actually slower than Maui without backfill even if only by a small amount. This can be explained by the job efficiencies, which for the test set of jobs had all but 5 jobs with an efficiency of 50% greater and 23 of the 76 jobs with an efficiency greater than 90%. If all jobs had an efficiency of 100% then the backfill algorithm would always be faster. However, since most jobs finish significantly before their scheduled end time it is possible that a backfilled job will keep a reservation from being started early. It is important to note that backfilled jobs will never prevent a job reservation from starting on time, but it might prevent a job reservation from moving forward in time.

Figure 2 illustrate the quite different ways in which the Maui Scheduler and the FIFO scheduler operate. The upper frame shows time, in hours, on the x-axis and job sequence number on the y-axis for the FIFO scheduler. The job sequence number represents the order in which the jobs were submitted to the system. The lower half shows the same information for the Maui Scheduler.

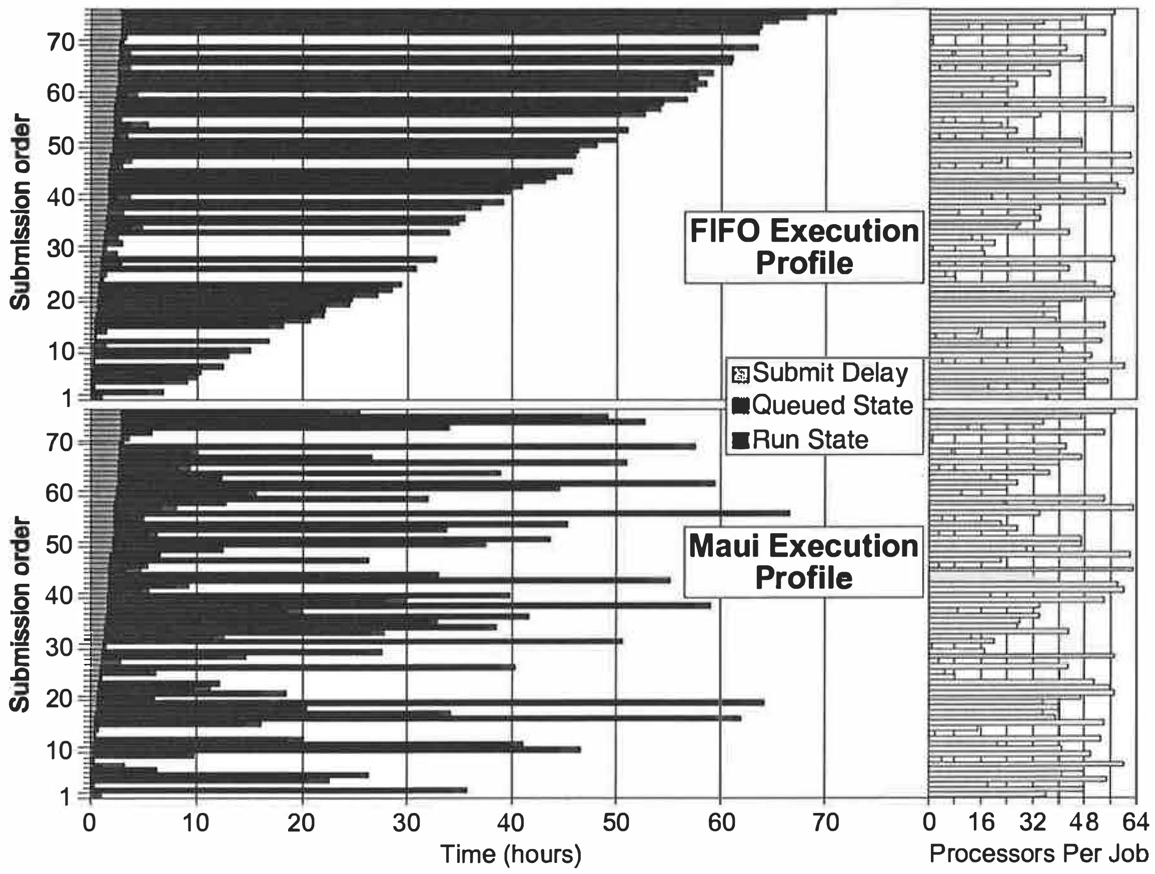


Figure 2. The execution profiles for the FIFO and Maui batch queues are presented in the left bar chart showing the submission delay, the wait time, and the run time respectively for each job. The right panel shows the number of processors requested by each of these jobs when they execute.

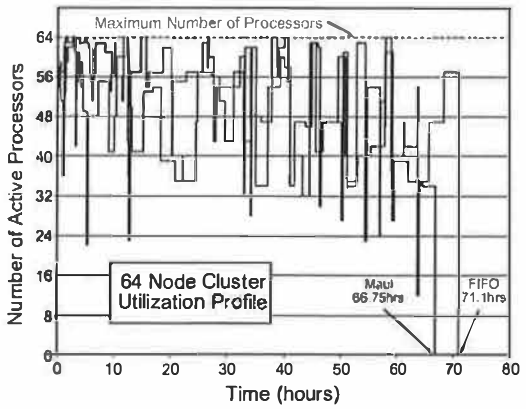


Figure 3. Cluster utilization comparison for PBS with FIFO, and with the Maui Scheduler active.

Since the Maui Scheduler result without the backfill algorithm was so similar to the regular Maui Scheduler result, it was not plotted separately.

For the FIFO scheduler the job profile shows that initially mainly small jobs were run up until the starving job state kicked in for the for the first large job in the queue. Once in the starving job state FIFO became truly a first in first out scheduler.

The profile for the Maui scheduler is certainly anything but FIFO. It shows a more uniform queue wait time that is driven more by the number of nodes requested than by the initial queue order. This results in the smaller node requests being run along with the larger node requests rather than all at once as with the FIFO scheduler. Because of this the Maui Scheduler is able to maintain higher average node utilization during the first portion of the test run, until it runs out of small node requests to backfill. This effect is illustrated in Figure 2 that shows the node utilization over the test run for all three scheduler tests. Figure 3 shows that Maui is able to maintain a more consistently high node utilization until about halfway through the test when it ran out of small jobs. The FIFO scheduler started out high, but then suffered a large dip as it cleared out the small jobs to let a large job start.

A further analysis of the data reveals that the FIFO scheduler starts all of the jobs with fewer than 10 nodes requested within 1 hour of submission. On the other hand Maui starts the last job with fewer than 10 nodes after over 16 hours in the queue. This difference is significant because it allows Maui to overlap the execution of more jobs during the test run, than does the FIFO scheduler. Indeed while the FIFO scheduler produces queue wait times nearly independent of the number of processors, ignoring the small jobs, the queue wait times under Maui are more similar to a bell curve with the maximum wait times experienced by jobs with node requests of approximately half the number of available nodes.

6.2 Theoretical Simulated Job Mix Results

In order to better evaluate the different schedulers performance for the evaluation job mix, a simulation routine was implemented that determined the first in first out (FIFO) execution of an ordered series of jobs. All jobs are executed in the specified order filling the system to the maximum number of nodes (e.g., 64). By running this routine with the submit order of the 76 simulation jobs, the FIFO execution time would be 69.63 hrs. Executing them in reverse order gives a FIFO execution time of 70.46 hrs. The jobs ordered as they were run in both the PBS/FIFO and PBS/MAUI simulations yields 69.16 hrs and 65.99 hrs, respectively. This demonstrates that the delayed submission does have an effect on how the jobs are eventually executed. In theory, with this routine we could find the optimal order for this specific set of jobs. Since there are 76 factorial (76!) possible orders, we did not pursue this.

6.3 Real Job Mix

The alpha cluster, running under moderately loaded conditions, has averaged 78% node hour utilization over the past three months. This was achieved while exceeding a total of over 1,200 jobs with node usage between 1 and 16 CPUs (Ave. of 4.3) and up to 2 wall days of time, the queue maximum. Out of the 1,200 jobs only 65 experienced a queue wait time of more than one day and most, 880, waited less than one hour to start execution.

This performance is despite the fact that the users are very poor at predicting the run time of their jobs. In fact the vast majority, 1146, of the jobs simply requested the maximum queue run time (2 days). The resulting job efficiencies were quite poor with only 140 jobs having an efficiency greater than 50% (i.e. using more than half of their requested time). It is unlikely that the job efficiencies will improve unless the load on the cluster increases producing longer queue wait times. Without long queue wait times users do not have much incentive to accurately predict the job run time or to attempt to fit a job into an existing hole in the job backfill window.

7. Future Directions

While we feel that the Maui Scheduler does an excellent job of scheduling jobs on flat interconnected clusters, a major area of on-going research is locality based scheduling. That is, scheduling based upon the topology of the interconnect, which might include interconnects with a tree structure and will certainly include SMP building blocks. This type of scheduling will become even more important in the near future since it becomes increasingly difficult and expensive to build a flat interconnect as the cluster size grows. In addition, new interconnect technologies are appearing which use loop, mesh and torus topologies.

There are of course many other areas of job resource management that need improvement on clusters. For example, job startup, monitoring and cleanup should be done in a parallel fashion. In addition the database of node and job status needs significant work to handle large clusters with large numbers of jobs effectively.

We plan to augment the simulations here with several techniques. First instead of a single pre-defined list of jobs randomly generated from a single source we will use "user-agents" that will submit jobs to the system. Each user-agent will submit jobs randomly generated but from a sub-class of the overall job mix. For example, a user-agent might represent a code developer submitting many debug jobs during normal working hours, a heavy user that submits long jobs, a greedy user that submits many jobs which fill gaps that a backfill mechanism might recognize, etc. The second modification is to change the metric. That will become the total number of active node hours in a given fixed time length. The user-agents will stop submitting jobs only after the metric has been met.

8. Conclusions

We have shown that the Maui Scheduler plugin to the PBS package provides a significant improvement in overall cluster utilization compared with the built-in FIFO scheduler. The Maui Scheduler does this by combining an intelligent wall time based node reservation system with an efficient backfill algorithm. The result is a flexible policy based scheduling system that provides guaranteed maximum start times while maintaining high total node utilization.
There are many issues that have yet to be addresses, such as cluster queue aggregation, inter-site trust and delayed authentication in addition to scalable system monitoring over a large distributed system.

9. References

[1] S. M. Jackson "QBank, A Resource Allocaiton Management Package for Parallel Computers, Version 2.6" (1998), Pacific Northwest National Laboratory, Richland, Washington 99352-0999.
[2] www.extremelinux.org/activities/usenix99/ docs/
[3] M. W. Schmidt, K. K. Baldridge, J. A. Boatz, S. T. Elbert, M. S. Gordon, J. H. Jensen, S. Koseki, N. Matsunaga, K. A. Nguyen, S. J. Su, T. L. Windus, M. Dupuis, J. A. Montgomery J.Comput.Chem. **14**, 1347-1363(1993)
[4] PBS_1, http://www.pbspro.com/
[5] http://www.mhpcc.edu/maui
[6] NERSC, The National Energy Research Scientific Computing Center,
http:/www.nersc.gov
[7] This complete data set is provided to allow the reader to reproduce our simulations if desired:
http://www.scl.ameslab.gov/Personnel/rickyk/ jobmix.html

SSH Port Forwarding

Giles Orr
Jacob Wyatt
Georgia College & State University

SSH Port Forwarding allows the use of the encrypted SSH tunnel between hosts to forward information on connections that would not normally be encrypted. Using this powerful tool is initially daunting, but is fairly straight–forward when it is understood.

1. Introduction

SSH stands for "Secure SHell." SSH is a replacement for telnet, rsh, and rlogin, to allow secure shell access to remote machines over an untrusted network. Telnet was designed at a time when the Internet consisted of a relatively small number of universities, and no one had ever heard of a packet sniffer. Packet sniffers such as sniffit and tcpdump are now relatively common – they have some highly practical uses, but obviously can also be used to collect passwords of those using unencrypted connections on a local network. Even if the password handshaking is encrypted, quite a bit of personal information can be collected watching an unencrypted transaction after the passwords. SSH packets looks like garbage to a packet sniffer.

2. Available Versions

SSH is currently available for free in several different versions – at least three versions for Unix, and at least two free ones for Windows. Version 1 of SSH is available for free for non–commercial use,*but is under a more restrictive license than the Gnu Public Licence.* It is maintained by SSH Communications Security Limited – although they intend to drop support for SSH 1 soon. Version 2 is being maintained and developed by the same people, including Tatu Yl˜ nen who originally wrote SSH – like SSH, it's available free for non–commercial and educational use, but the license is still not GPL.

There is also the OpenSSH project currently under way. It was developed by the OpenBSD people, under the OpenBSD licence. OpenSSH has port forwarding with the same command line as SSH 1, although X forwarding is disabled at installation for security reasons with the RPM packages of OpenSSH that we worked with. Most versions also rely on OpenSSL, so you should have that installed before you try to install OpenSSH.

In the Windows world, you can use TTSSH or PuTTY to connect to an SSH server. TTSSH supports port forwarding both from the GUI and the command line. PuTTY doesn't seem to support forwarding at all yet, but it's still beta. There are also pay versions of SSH for Windows available (primarily from F–Secure, who are directly associated with SSH Ltd.), but we won't be discussing these. We will be addressing SSH v1 on Unix, and to some extent connecting to Unix from Windows clients.

SSH is available from ftp.ssh.com:/pub/ssh as a tarball. *RedHat and other users of rpm packages might be able to get ssh and openssh from rpmfind.net, but major distribution vendors don't seem to be contributing, so make sure you trust the source of the package.* In the case of cryptography packages like this one, they used to link to www.replay.com. Replay is now Zedz.net, and rpmfind no longer links to them. To the best of our knowledge, Zedz.net's service is still sound: you can find packages at ftp://ftp.zedz.net/pub/crypto/redhat/i386/ . Binary packages are available for other distributions as well. If you're really serious about your cryptography, you'll get the sources, check the PGP signature, check the source for backdoors, and compile it yourself. However, that's a fairly arduous task, and not what we're here to discuss.

3. Basic Use of SSH

The most basic use of SSH is as a replacement for telnet and rsh. At a command prompt, just type "ssh hostname.com" instead of "telnet hostname.com". This will only work if hostname.com has the SSH server software installed. We encourage you, if you're a system administrator, to turn off your telnet servers, and switch completely over to SSH.

Login can be set up in two ways, either using a PGP key, or plain password. Either way, SSH never passes anything in the clear: two way handshaking and exchange of session crypto keys takes place before any passwords or passphrases are sent. After you send your password, the session behaves more or less like telnet, but it's encrypted at all times. SSH behaves in the same manner as rsh, in that it assumes your login name on the remote server is the same as on the local one. To override this behaviour, use something like this:

```
root@localhost$ ssh remotehost -l
giles
```

4. Port Forwarding

The concept of port forwarding is relatively simple. Unfortunately, the command line that accompanies it tends to be a little cryptic, and there are some twists and turns to work through. The idea is to allow any connections made to a port on the local machine to be sent through the encrypted SSH connection to the remote machine, so connections that would normally be clear text (such as FTP, SMTP, or NNTP, just to name a few) can benefit from SSH security.

5. Forwarding X

One of the great beauties of SSH is that X Forwarding can be (and usually is) enabled during compilation, so that after you have connected to another server with:

```
giles@tesla$ ssh remotehost

giles@remotehost's password:

No mail.

giles@remotehost$
```

You can start X clients simply by typing them as you would at the console of the remote machine, and they will appear on your local machine:

```
giles@remotehost$ xclock &

[1] 26416

giles@remotehost$ xterm &

[2] 26422

giles@remotehost$
```

This is because SSH has already set up your local machine as the machine to display programs on:

```
giles@remotehost$ set | grep DISPLAY

DISPLAY=tesla.gcsu.edu:10.0

giles@remotehost$
```

If you want to start using your local machine as the work environment for your remote host, one of the cooler things you can do is start an application manager like the Gnome project's "panel", or perhaps something like "tkdesk" or "xfm" if that's more your style. If you're not familiar with doing something like this, a word of warning: you have to be cautious about remembering where any given app has been launched from, either locally or remotely. If you're reconfiguring the remote host, and you type "rm –rf /usr/local/bin/*" on the wrong machine, it will be very painful ... One solution for that is to colorize your prompts differently on every machine you work on, with the host name prominently displayed.

Often X forwarding is turned off on servers that have a lot of users. My ISP in Toronto, for example, has it turned off, presumably because their business is dial–up connectivity and web hosting, so they don't want the machine bogged down by fifty users all running Netscape on their processors. I tend to turn it on on my servers, but they usually only have twenty or thirty users, and most of those people are only using FTP.

6. Forwarding News

Now let's move on to the forwarding that you can do that isn't configured for you. Perhaps the easiest way to explain what this is about is to give an example that happened to me.

I live in Georgia, but I'm originally from Toronto. I'm still interested in what happens in Toronto, so I like to look at the Toronto news groups occasionally. So I set up port forwarding for NNTP, Network News Protocol, to allow me to read the news groups at my ISP in Toronto rather than on my local ISP, which doesn't carry Toronto groups. NNTP is carried on port 119, and you use some kind of news reader to look at it. I usually use trn – a bit arcane, but quite powerful.

The servers we're dealing with in this case are "shell.interlog.com" and "news.psi.ca". The former is my ISP's shell server, and the latter is their news server. I can't log in to the news server directly, it has no shell access. I can read news by logging in on shell.interlog.com and running trn, but I'd rather do that on my home machine. And since I can't make news requests from my home machine (it's not in the IP range news.psi.ca accepts connections from), I had to figure out how to make it work another way. This looks a bit daunting, bear with me.

```
giles@tesla$ su -c "ssh -C -L
    119:news.psi.ca:119
    shell.interlog.com -l giles"
```

I'm going to look at this starting at the back end. Since I'm using su to become root (I'll explain why in a second), ssh thinks I'm root on the remote machine. I don't have root on my ISP, so I have to tell it otherwise. Since I can't connect directly to news, I connect to shell. It isn't the machine that I'm getting news from, and that's one of the interesting twists here. Since "max3–42.dial.accucomm.net" (my assigned dial–up IP address) isn't authorized to get news from news.psi.ca, we connect to a machine that is authorized. I connect and port–forward through shell.interlog.com, which I have access to, and which is authorized to get news from news.psi.ca. So the "–L 119:news.psi.ca:119" is telling ssh to take port 119 on the local machine and send all information from there to port 119 on news.psi.ca. shell.interlog.com acts as a relay. The "–C" flag is important for me at home: I have a 56k dial–up connection, and "–C" means "use compression." This isn't useful if you have ethernet because the overhead of the compression slows you down more than the compression speeds you up, but on a phone line, it's great. Note that the connection isn't encrypted between shell and news, but that's not as important as having it encrypted on the Internet at large.

Finally, why did I su to root? Port 119 is a privileged port. On the remote machine, I'm only feeding data to it, so I don't need root, but on the local host, I'm taking it over entirely, so I have to be root. All ports below 1024 are privileged, so you have to have root to forward them. Often, if you don't have root, you can use another port above 1024, and tell the application you're using to look at the other port you chose – when you use "–L" with SSH, the two port numbers DO NOT have to be the same.

Now I'm ready to read news on my machine, as if I was directly connected to my ISP in Toronto. You need to set one environment variable on the local machine, and you're ready to go:

```
giles@tesla$ NNTPSERVER=localhost ;
    export NNTPSERVER
giles@tesla$ trn
```

That's it: you should be up and running.

7. Forwarding FTP

I recently put my web pages on another ISP in Toronto. Since I live in Georgia, my local ISP is quite a few hops away from my ISP in Toronto, and there are a lot of computers I don't trust between me and my new web host. So I use ssh to connect to them, and I would use scp (a secure replacement for rcp that's usually bundled with ssh) to copy my files to and from their service. Unfortunately, for reasons unexplained, they don't have scp (secure copy, a part of the SSH package) installed. So I decided to use port forwarding to use FTP to connect to them. In this case, it was a bit easier to do it as a user rather than as root:

```
giles@tesla$ ssh -C shell.eol.ca -L
    2121:ftp.eol.ca:21
giles@shell.eol.ca's password:
    Warning: Remote host denied X11
    forwarding, perhaps xauth program
    could not be run on the server
    side.
giles@eol$
```

Note the warning: you will see something like that when X11 forwarding is turned off. Now, in another window, I do something like this:

```
giles@tesla$ ftp localhost 2121
Connected to localhost.
```

```
220 babbage.echo-on.net FTP server
(BSDI Version 7.00LS) ready.
Name (localhost:giles): giles
331 Password required for giles.
Password:
230 User giles logged in, access
    restrictions apply.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp>
```

As soon as you issue that ftp command you'll see a message in your other window that's connected to the remote machine:

```
giles@eol$ fwd connect from 127.0.0.1
    to local port sshdfwd-2121
```

At this point, it looks like everything is cool and you're in business. But ...

```
ftp> ls
200 PORT command successful.
425 Can't build data connection:
    Connection refused.
ftp>
```

The problem is that FTP spawns connections on other ports by default to take care of the work. Since the other ports aren't forwarded, it tends to get confused. This is the biggest "gotcha" I've found so far in SSH port forwarding, it took quite a while to figure this out.

So I issue the following command:

```
ftp> passive
Passive mode on.
```

```
ftp> ls
227 Entering Passive Mode
(205,189,151,4,203,122)
150 Opening ASCII mode data
connection for '/bin/ls'.
total 134
...
```

If you're using a GUI client, you may need to do some digging to figure out how to convince it to switch to passive mode. My preferred client these days is lftp, and I found out by digging through the man page that what I needed to do was this:

```
giles@tesla$ lftp -p 2121 -u giles
localhost
Password:
LFTP giles@tesla:~
> set ftp:passive-mode on
```

It would also be relatively easy to put a setting in your ~/.lftp/rc file so that all connections to local host were in passive mode by default.

Another way to establish the same connection would be to use the following command line:

```
ssh -f -C -L 2121:ftp.eol.ca:21
    shell.eol.ca "sleep 30"
```

There are only two differences here: the "–f" flag and the command "sleep 30" added on the end. "–f" requests ssh go to background after authentication is done and forwardings have been established. This will get you your terminal back, and "sleep 30" allows you enough time to set up a connection with ftp to keep the forwarding alive. It will die when you disconnect.

8. Port Numbers

You may be wondering how I know what port numbers to use. This is actually surprisingly easy to find out: "less /etc/services" *on a UNIX machine* will tell you a great deal about what ports are used for what purposes. I'll outline a couple more examples, but when we're done, if I haven't covered what you want to do, you should have the tools to figure it out yourself.

9. Forwarding Mail under Windows

Jacob and I work at a mid–sized university in middle Georgia, and all of the university dorms are wired. The university also has both a Computer Science program and a Business Information Systems program, so we have a lot of potential crackers live on our network before you even consider the fact that we have a T1 to the rest of the internet without a firewall ... So when I collect my mail using Eudora running on Windows, I'm not enthusiastic about having my password sent in clear text three or four times a day, or however often I press the "Check Mail" button. So I used a free Windows SSH client to secure my connection to our mail server – fortunately, we're allowed telnet and ssh access. The mail server administrator has set it up so that as soon as you log in, Pine starts up, and there's no way out to the shell (without getting very creative), but this doesn't matter: forwarding is taken care of on your local machine, so long as an SSH connection is established.

The program that I use is TeraTerm Pro, a very nice terminal emulator for Windows. It's available at http://hp.vector.co.jp/authors/VA002416/teraterm.html . You will also need the SSH extensions, available at http://www.zip.com.au/~roca/ttssh.html , which turn TeraTerm into an SSH client. Both of these are available for free. You can set port forwarding up in the "Setup" –> "SSH Forwarding" menu item, or you can do it from the command line in batch mode:

```
"C:\Program Files\TTERMPRO\ttssh"
    mail.gcsu.edu:22 /ssh
    /ssh-L110:mail.gcsu.edu:110
    /ssh-L25:mail.gcsu.edu:25
```

I then have to give a password login to the mail server. Since it's Windows, I don't have to give a root password to forward privileged ports.

The first /ssh switch tells ttssh to run in SSH mode. The next two are each just like the –L switches to ssh under Unix. I'm forwarding SMTP (port 25) and POP3 (port 110), so both sending and retrieving mail is encrypted. Sending mail this way isn't of much benefit if the mail is going out into the world, because it goes clear–text as soon as it's past the mail server, but the majority of my mail is going to recipients on the same mail server.

10. Pitfalls

Once SSH Forwarding is established on a Unix box, all users of that machine can – and in fact, have to – use the forwarding. If someone is trying to FTP to localhost (perhaps to access files as another user) while you have forwarding set up on that port, they'd get a big surprise, finding themselves connected to another server. This would be a pretty good argument for using a high number non–standard port if you're working on a multi–user system. However, it would occasionally be advantageous to allow other users on other machines to use a forward that you have set up: by default, SSH doesn't allow this, but you can switch it on if you like. The option is "–g". I would suggest using this with extreme caution.

11. Conclusion

SSH Port Forwarding is primarily a single–user pursuit, and any access by multiple users should be approached with considerable caution. Initial setup can also be a bit tricky, and we recommend that you create scripts to assist you in recreating the forwards you need. Within these limitations, SSH Port Forwarding is an extremely useful extension to the security provided by SSH, allowing you to encrypt many other applications that didn't originally present a security risk in an unencrypted form.

12. Resources

SSH Communications Security Limited is located at http://www.ssh.com/ . Source code for either version of SSH is available. You can also go directly to ftp://ftp.ssh.com/pub/ssh/ .

http://zedz.net/ has RPM and source packages for most of the Unix software discussed in this talk.

OpenSSH can be found at http://www.openssh.com/portable.html.

Putty can be obtained at http://www.chiark.greenend.org.uk/~sgtatham/putty/ : it is a beta Windows–based SSH client. At the time of writing, it doesn't support port forwarding, but it does include a Windows version of scp called pscp.

TTSSH is an add–on for Teraterm. Teraterm is an excellent Windows software terminal emulator, available at http://hp.vector.co.jp/authors/VA002416/teraterm.html . TTSSH is available at http://www.zip.com.au/~roca/ttssh.html . Commonly used ports (which you can check for yourself in /etc/services on most Unix machines) are given below.

```
ftp-data      20/tcp
ftp           21/tcp
ssh           22/tcp                       # SSH Remote Login
telnet        23/tcp
smtp          25/tcp    mail
www           80/tcp    http               # WorldWideWeb HTTP
pop-2         109/tcp   postoffice         # POP version 2
pop-3         110/tcp                      # POP version 3
nntp          119/tcp   readnews untp      # USENET News Transfer
imap2         143/tcp   imap               # Interim Mail Access
snmp          161/udp                      # Simple Net Mgmt Proto
irc           194/tcp                      # Internet Relay Chat
imap3         220/tcp                      # Interactive Mail Access
exec          512/tcp
biff          512/udp   comsat
login         513/tcp
who           513/udp   whod
shell         514/tcp   cmd                # no passwords used
syslog        514/udp
printer       515/tcp   spooler            # line printer spooler
talk          517/udp
ntalk         518/udp
uucp          540/tcp   uucpd              # uucp daemon
rsync         873/tcp                      # rsync
mysql         3306/tcp                     # MySQL
ircd          6667/tcp                     # Internet Relay Chat
```

Enhancements to the Linux Kernel for Blocking Buffer Overflow Based Attacks

Massimo Bernaschi[1] Emanuele Gabrielli[2] Luigi V. Mancini[2]

[1] Istituto Applicazioni del Calcolo, CNR
Viale del Policlinico 137, 00161 Roma, Italy
massimo@iac.rm.cnr.it

[2]Dipartimento di Scienze dell'Informazione
Università di Roma "La Sapienza", 00198 Roma, Italy
gabriell@dsi.uniroma1.it, lv.mancini@dsi.uniroma1.it

Abstract

We present the design and implementation of a cost-effective mechanism which controls the invocation of *critical*, from the security viewpoint, system calls.

The integration into existing UNIX operating systems is carried out by instrumenting the code of the system calls so that the system call itself once invoked checks to see whether the invoking process and the argument values passed comply with the rules held in an access control database.

A working prototype able to detect and block buffer overflow attacks is available as a small set of "patches" to the Linux operating system kernel source.

1 Introduction

We propose an approach to the control of system calls which requires minimal additions to the kernel code and neither changes to the syntax and semantics of the system calls nor modifications of existing kernel data structures and algorithms. All kernel enhancements are transparent to existing or new programs. No change in the source code or special compiling procedure is required.

Basically when *controlled* system calls are invoked, our mechanism checks whether the invoking process and the value of the arguments comply with the rules kept in an Access Control Database (ACD) placed within the kernel.
To reduce the cost of the checks, a detailed analysis of privileged applications and system calls is carried out. This allows to identify the set of primitives which may be dangerous for the system security.

As an example of this methodology, we have designed and implemented a prototype which prevents privileged processes from compromising the security and integrity of the Linux operating system when they are subverted by means of techniques like *buffer overflow*.

Buffer overflow is a widely known technique [AlephOne] which allows to force a process generated by a buggy program in executing "fake" instructions injected by the attacker. If the technique is successfully applied to a privileged process and the fake code is used, for instance, to start the execution of an interactive shell, the attacker gains the access to a privileged shell.

During the design phase, the complete set of Linux system calls has been analyzed. The result of the analysis shows that by adding access control tests to a small number of system calls, the protection against buffer overflow is complete and can not be bypassed by executing unprotected system calls. This reduces the cost of system call interception since the invocation of most system calls is not checked.

Any process running with root privileges is a potential target of a buffer overflow attack. Our control mechanism prevents these processes from executing unexpected system calls if they undergo an attack.

2 Problem analysis

2.1 Privileged processes

For the purpose of our discussion, a privileged process may belong to one of the following three categories:

interactive: This is a generic process started by the system administrator. Both the User IDentifier (UID) and the Effective User IDentifier (EUID) are equal to 0. It does not make much sense to monitor such processes, since any user able to start them has the full control of the system. However we must prevent a privileged process from migrating to this category if it was started in a different one.

background: This is, usually, either a daemon process started at boot time or a process started periodically by the cron daemon on root behalf. Following [Stevens] and [Comer] we assume that such processes *never* need a control terminal. To distinguish them within the kernel, we resort to the following check:

```
!((proc)->euid)&&((proc)->tty==NULL)
```

Here, the first logical clause checks whether the process runs with root privileges (EUID=0) whereas the second one checks whether the process has a controlling terminal. We block any attempt made by these processes to re-acquire a control terminal. Note that a daemon can still open a terminal device (e.g. `/dev/tty` or `/dev/console`) to log error messages.

setuid: When a program with *setuid* access mode is executed, the effective UID of the process is set equal to the UID of the program file owner. As a consequence, the access to files and system resources is carried on with the identity of the owner of the program file. This is the standard UNIX mechanism to grant ordinary users with special privileges on a temporary basis. A process can be identified as setuid to root (EUID=0) by means of the following simple check:

```
!((proc)->euid)&&(proc)->uid
```

Note that a setuid process started by the user *root* has $UID = 0$. For this special case the same considerations made for an interactive root process apply.

2.2 System calls analysis

A drastic technique to prevent the damages produced by buffer overflow attacks would be to abort the execution of all system calls invoked by non-interactive root processes. However, the abort of all system calls is not possible (root processes can legally invoke some system calls), and is not needed (many system calls are not dangerous even when invoked via a buffer overflow attack).

The system calls available in Linux 2.2 have been grouped in categories according to their functionality as reported in table 1. In addition, each primitive has been classified according to the level of threat that it may introduce following the definition reported in table 2.

Currently we address the issues raised by the system calls classified as *threat* level 1 which are reported in table 3.

For different reasons no system call in the groups IV-IX can be used to gain the control of the system. For instance system calls in group IX return immediately the error `ENOSYS`, whereas primitives in group VIII can not be invoked by a generic user process (even if it is a privileged process).

As to the system calls in group III, a subverted process may use them for loading a *malicious* module. Our investigation shows that `create_module` is the only primitive which reaches the threat level 1 since no module can be activated without invoking `create_module`. The term *malicious* in table 3 has a very broad meaning. Basically we block any module which is not listed in the ACD.

Among the primitives related to process management, ten reach the highest level of danger for the system security. The `execve` can be used to start a root shell. The other nine primitives set user and group identifiers. It is worth noting that `capset` allows a process only to restrict its capabilities, so it belongs to class 3.

I	File system and devices	V	Communication
II	Process management	VI	Time and timers
III	Module management	VII	System information
IV	Memory management	VIII	Reserved
		IX	Not implemented

Table 1: System calls categories

threat level	threat description
1	May be used to get full control of the system
2	May be used for a denial of service attack
3	May be used for subverting the behavior of the invoking process
4	It is harmless

Table 2: Threat level classification

system calls	dangerous parameter
`chmod, fchmod`	a system file or a directory
`chown, fchown, lchown`	a system file or a directory
`execve`	an executable file
`mount`	on a system directory
`rename, open`	a system file
`link, symlink, unlink`	a system file
`setuid, setresuid, setfsuid, setreuid`	UID set to zero
`setgroups, setgid, setfsgid, setresgid, setregid`	GID set to zero
`create_module`	a malicious module

Table 3: Threat level 1 system calls

The ten system calls related to the file system classified as threat level 1 require special attention. It is apparent that the execution of

```
chmod(''/etc/passwd'',0666)
```

compromises the OS authentication mechanisms. However, it is necessary to consider *chains* of system calls as well. For instance, chown and chmod primitives can be used in a two-steps procedure to create a setuid shell, whereas the following sequence:

```
unlink(''/etc/passwd'')
link(''/tmp/passwd'',''/etc/passwd'')
```

produces the same result of the `rename` primitive.

Moreover, by means of a buffer overflow, it is possible to execute code that adds to the `/etc/passwd` (and/or `/etc/shadow`) file a new user with UID=0 or adds a fake *.rhosts* file to the root home directory. Although less common, these exploits are, by no means, less dangerous than those based on the classic *shellcode*. Most attacks that involve a file require at least two system calls. A first one to open the file and a second one to modify it. In this case, in accordance with [Goldberg] we assume that it is necessary to monitor just the open primitive. That is the reason why the `write` system call is not considered threat level 1.

In building an ACD for the open primitive a number of different situations must be considered. Several setuid programs or root daemons may open, for good reasons, critical files (e.g., `/etc/hosts.allow`). For the time being, we assume that opening such files in read-only mode is harmless. We understand that it may be possible to steal precious information like the encrypted passwords or the names of "trusted" hosts but we are going to address such issues in a second time. We recall that the checks are enforced on setuid or daemon programs just in case they invoke critical system calls with root privileges (that is with $EUID = 0$). Since most of the times these programs give up to their privileges (the EUID is set equal to the real UID) before opening files in write mode, the access to user files does not need to be authorized. A detailed study of setuid and daemon programs has been carried out to define which directories and files must be included in the ACD. This has been realized by both source code inspection and analysis of the results produced by tools like the `strace` command which intercepts and records the system calls invoked by a process.

```
/* execve_acd */

typedef struct setuid_proc_id {
        char comm[16];
        unsigned long count;
} suidpid_t;

typedef struct  setuid_program {
        suidpid_t suidp_id;
        suidp_t   * next;  /* next program */
} suidp_t;

typedef struct exe_file_id {
        __kernel_dev_t   device; /* device number     */
        unsigned long    inode;  /* inode number      */
        __kernel_off_t   size;   /* size              */
        __kernel_time_t  modif;  /* modification time */
} efid_t;

typedef  struct executable_file {
        efid_t   efid; /* info for file identification */
        int prog_nr;
        /* number of programs that can invoke this exe */
        suidp_t  *programs; /* list authorized programs */
} efile_t;

typedef struct executable_file_list  {
        efile_t  lst[NR_EXE];
        unsigned int total;
        /* total number of exe in the list */
} eflst_t;
```

Figure 1: The layout of the `execve_acd` data structure

3 The Access Control Database

The Access Control Database contains a section for each system call kept under control. For instance, the working prototype maintains a `setuid_acd` data structure to check the access to the `setuid` system call, and an `execve_acd` data structure to check the access to the `execve` system call. The layout of `execve_acd` data structure is shown in Figure 1:

The execve_acd is composed by two arrays of `eflst_t` structures:

admitted: an executable file F has an entry in this structure if, at least, one privileged program needs to execute F via an `execve`. The information stored in the entry is the list of all

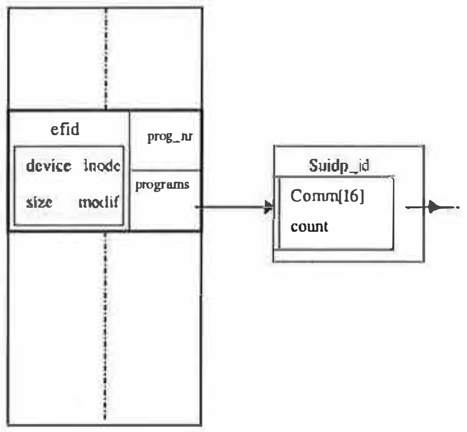


Figure 2: The layout of the *admitted* data structure

privileged programs which may invoke F.

failure: this list keeps a log of the unauthorized attempts (that is, not explicitly allowed by the `admitted` data structure) of invoking `execve` by any setuid process.

Figure 2 shows the `admitted` data structure which is an array where each element refers to an executable file and points to a list of setuid programs that can execute that file.

Each element of the admitted data structure contains these fields:

`efid`: identifies the executable file F. The information stored in efid are:
`device` that is the device number of the file system to which file F belongs;
`inode` that is the inode number of file F;
`size` that is the length in byte of file F;
`modif` which keeps the last modification time of file F.
The pair of information `device` and `inode` identifies file F in a unique way within the system whereas the information `size` and `modif` allow to detect unauthorized modifications of the file contents.

`programs`: is a pointer to the list of privileged programs which can execute file F.

We have introduced a new system call `sys_setuid_aclm` for reading and modifying the information kept in the ACD.

This system call can be invoked only by interactive root processes with EUID=0 and UID=0. These constraints are required to prevent a subverted setuid or root daemon from tampering the ACD.

The system administrator (root user) can manage the ACD resorting to the `sys_setuid_aclm` system call through a new command, named `aclmng` which offers the following options:

-l lists the contents of the Access Control Database kept in kernel space;

-L loads in kernel space the Access Control Database from file `/etc/bop/acd`, mostly used during booting;

-w writes the Access Control Database from kernel space into file `/etc/bop/acd`, mostly used during system shutdown;

default with no options, -l is assumed.

4 The reference functions

This section presents some examples of the extensions introduced to control the system calls defined in section 2.2 as threat level 1.

execve As shown in Figure 3, the new fragment of code is added at the beginning of this system call right after the file has been opened. The `check_rootproc()` function authenticates the privileged process that invokes the `execve` system call and checks in the Access Control Database whether the calling process has the right to execute the program whose name is passed as first parameter. The system call execution is denied when `check_rootproc` returns one of the two following values:

EXENA: the calling process is not authorized to execute the requested program. That is, the program name is not present at all in the Access Control Database or the calling program is not listed in the `programs` field of the admitted list in the Access Control Database.

EFNA: the calling process is authorized to execute the requested program, but the file is not authenticated, e.g. the modification time or the size do not match.

```
/*
 * sys_execve() executes a new program.
 */
int do_execve
    (char * filename, char ** argv, char ** envp, struct pt_regs * regs){

    ...
    dentry = open_namei(filename, 0, 0);
    retval = PTR_ERR(dentry);
    if (IS_ERR(dentry))
      return retval;
    ...
    retval = prepare_binprm(&bprm);

    /*********** BUFF_OVERFLOW PATCH  ***************************/
    rc=check_rootproc(bprm.dentry->d_inode);
    if ((rc==EXENA) || (rc==EFNA)) {
      printk(BOP_LEVEL"BOP kernel:do_execve psuid %s no
             authorized to exec file %s\n",current->comm,filename);
      printk(BOP_LEVEL" by euid %d uid %d\n",
             current->euid,current->uid);
      if (rc==EXENA)
        printk(BOP_LEVEL" EXE NO AUTHORIZED\n");
      else  printk(BOP_LEVEL" EXE NO AUTHENTICATED\n");
      return rc;
    }
    /*************************************************************/
    ...
    if (retval >= 0)
      retval = search_binary_handler(&bprm,regs);
    if (retval >= 0)
       /* execve success */
        return retval;
    ...
}
```

Figure 3: The “patch” to the execve system call

In the appendix we show the details of the `check_rootproc` function. If the calling process does not run with root privileges (EUID=0) then no further check is performed and the execve proceeds normally. Otherwise, the service is provided if and only if the permission is explicitly contained in the Access Control Database.

setuid For the `setuid` system call, the authentication of the root processes is the same as in the `execve` case. A user running a setuid program which attempts to invoke `setuid(0)` to set the (real) UID equal to 0, is enforced to type the root password. The password keyed is compared with the encrypted copy kept in the Access Control Database. In case of a password mismatch the `setuid(0)` invocation is denied. So far only the program `su` (a setuid program which runs a shell with substitute user and group ID) needs to be monitored with this mechanism.

chmod The major difference with respect to the `setuid` primitive is that no root password is required whereas an additional check is performed on the `filename` argument. If `filename` refers to a regular file or a directory, the operation is denied. This means that the operation is allowed if `filename` refers to a device registered in the ACD. As usual if `chmod` is invoked by an ordinary user process or an interactive root process no additional check is done.

5 Installation and Performance

The software prototype (for availability see section 8) is composed of three parts:

- A kernel patch. The patch has been developed and tested with the version 2.2.12-20 of the Linux kernel. Since it is basically a set of additions to the existing code for the system calls (there are neither changes nor deletions), we do not expect major problems in porting it to other (newer) versions of the kernel.
- The new command `aclmng`
- A modified version of the `chmod` command. The only difference with the original program is that `chmod` accepts new options to add entries in the ACD. For instance, when `chmod` is used to add the setuid functionality (mode *+s*) to the *foo* program, owned by root, the *-p* option allows to specify the list of the programs that *foo* will be allowed to *exec*.

  ```
  chmod +s -p /program1:... :/programN foo
  ```

  allows the setuid program *foo* to execute any of *program1,... programN* (the execution of any other program is, by default, forbidden). What the modified `chmod` does is to invoke the `sys_setuid_aclm` system call to add the necessary information in the ACD.

The system administrator's duties are limited to run the new version of the `chmod` command. Neither recompilation nor code inspection is required. Messages sent to the syslog by the modified commands and by the system calls, start with the "BOP" prefix to spot them easily.

A very limited degradation of the global performance is expected for a system running our patched kernel. There are a number of reasons for this forecast:

- When a process runs in *user* mode, there is no difference at all with a standard system since all new checks are confined in the kernel.
- Very few primitives include new checks (approximately 10% of the total number of system calls).
- Only a limited subset of the processes execute all the checks.
- With the exception of the `open` primitive, it is unlikely that a setuid or daemon process invokes any of the instrumented system calls more than once during its lifetime.
- The checks do not require any access to "out of core" data, all the info is resident in the kernel memory.
- There are no large data structures, so the lookup is fast without requiring complex algorithms. For instance, the number of setuid programs which need to *exec* other programs is less than five in a typical Linux configuration.

To assess these considerations, a set of experiments has been executed. We have selected four applications and ran them on the same system (a 330 MHz

Pentium II with 128 MB of RAM) with a standard Linux kernel (version 2.2.12) and the same kernel "patched" to include the additional checks. Each test has been repeated 40 times. The applications have been used as follows: `sendmail:` by means of a simple shell script three messages of different size (1 KB, 30 KB and 1 MB) have been sent to a local user;
`lpr:` 8 files of different size (from 1 KB to 10 MB) have been sent to a local printer;
`rsync:` a directory with 1440 files (total size about 10 MB) has been synchronized with a different path (on the same system);
`X server:` by means of the `x11perf` program a 300×300 trapezoid is filled with a 8×8 stipple.

It is apparent looking at the results reported in table 4 that the difference between the average execution times is comparable with the standard deviation of the multiple runs. This confirms that the actual impact of the *patches* on the global system performance is, for all practical purposes, negligible.

6 Related work

Buffer overflow based attacks have been around, at least, since 80's and many solutions have been proposed in the past to solve the problem *definitely*. We are not going to offer here an exhaustive review of all possible approaches. We list just some solutions recently proposed.

Marking both *data* and *stack* regions as non executable may catch most "cut and paste" exploits. A non executable stack is readily implemented [Solar] since it introduces just minor side effects in most UNIX variations (e.g., Linux places code for signal delivery onto the process's stack). Note that there is no performance penalty and existing programs require neither changes nor re-compilation (unless they use exotic features like gcc *trampolines*). The situation is not so simple for the data region. It is not possible to mark it as non-executable without introducing major compatibility problems. Even if this could be solved, there is still the problem of attacks which instead of introducing *new* code, corrupt code pointers. This technique allows to execute dangerous instructions which are already part either of the program or of its libraries [Wojtczuk].

Compiler techniques have been proposed for introducing in an executable code "lightweight" checks on the integrity of functions' return address.

StackGuard [StackGuard] detects and defeats stack smashing attacks by protecting the return address on the stack from being altered. StackGuard places a "canary" word next to the return address when a function is called. If the canary word has been altered when the function returns, then a stack smashing attack has been attempted, and the program responds by emitting an intruder alert into syslog, and then halts.

The major limitation of StackGuard is that it protects against buffer overflows *in the stack*. Unfortunately, heap overflows are less common but, by no means, less dangerous than stack overflows [Conover]. Moreover it has shown very recently that it is possible to exploit buffer overflow vulnerabilities in the stack even in programs compiled with StackGuard or StackShield [Bulba], [Bouchareine]

Other groups in the past have proposed to address security issues by means of special controls on the values of system calls arguments. In [Goldberg] a user–level tracing mechanism to restrict the execution environment of untrusted helper applications is described. Our solution is based on a similar (but independent) analysis of the potential problems associated with *some* system primitives, but we control a different set of programs (i.e. root daemons and setuid programs instead of helper applications). We add our additional checks to the system calls code mainly for performance reasons but the impact on existing kernel code is reduced to the bare minimum (*no* change just additions).

The Domain and Type Enforcement (DTE) is an access control technology which associates a *domain* with each running process and a *type* with each object (e.g, file, network packet). At run time a kernel-level DTE subsystem compares a process's domain with the type of any file or the domain of any process it attempts to access. The DTE subsystem denies the attempt if the requesting process's domain does not include a right to the requested access mode for that type. DTE is a very general approach to mandatory access control, however it requires deep kernel modifications (about 17,000 lines of kernel resident code) and 20 new system calls for DTE-aware applications [Badger].

More recently, a high level specification language called Auditing Specification Language has been in-

Table 4: Results from performance tests. We report the average execution time (in seconds) and the standard deviation of 40 runs

Application	elapsed time (standard kernel)	elapsed time (*patched* kernel)
sendmail	1.32 ± 0.05	1.33 ± 0.04
lpr	2.08 ± 0.1	2.1 ± 0.15
rsync	10.36 ± 0.8	10.56 ± 0.6
X server	0.101 ± 0.001	0.102 ± 0.002

troduced [Sekar] for specifying normal and abnormal behaviors of processes as logical assertions on the sequence of system calls and system call argument values invoked by the process. Unfortunately, not enough information are available up to now about their "System Call Detection Engine".

7 Future activities

We have described how *simple* enhancements of an existing kernel code can make harmless a well known threat for the system security like buffer overflow based attacks. Our prototype kernel has been in production for eight months in our organizations and so far no fault due to our patches has been reported by the users.

In the short term, we expect to add "reaction" capabilities to our attack detection mechanism. The starting point is to develop a kernel subsystem to manage intrusion attempts. Simple systems have been already used in the past to analyze the intruders' activities in progress without let them notice it. However those systems were not activated on the fly during the intrusion attempt. The real-time intrusion handling mechanism we have in mind requires the migration of the offending process to a distinct system designed to reproduce the original environment as faithful as possible. We are currently investigating which is the best way to implement this technique.

8 Availability

The prototype and an updated version of this document is available from:
`http://www.iac.rm.cnr.it/newweb/tecno/indexsecurity.htm`

References

[AlephOne] Aleph One,*Smashing The Stack For Fun And Profit*, Phrack Mag., V. 7, N. 49, 1996.

[Cowan] Cowan, C. *et al.*, *Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade*, to appear as an invited talk at SANS 2000, `http://www.cse.ogi.edu/DISC/projects/immunix.`

[Conover] Conover, M. and the *w00w00* Security Team, *w00w00 on Heap Overflows*, `http://www.w00w00.org/articles.html`

[Solar] Solar Designer, *Non-Executable User Stack* `http://www.openwall.com/linux`

[Wojtczuk] Wojtczuk R., *Defeating Solar Designer Non-Executable Stack Patch*, Bugtraq mailing list: January 30 1998.

[StackGuard] StackGuard: `http://www.cse.ogi.edu/DISC/projects/immunix/StackGuard`

[Stevens] W.R. Stevens, *Unix Network Programming*, II edition, Prentice Hall 1998.

[Comer] D. E. Comer and D. L. Stevens, *Internetworking with TCP/IP Volume III*, Prentice Hall 1998.

[Goldberg] I. Goldberg, *et al.* "A Secure Environment for Untrusted Helper Applications", Proceedings of the USENIX 6th UNIX Security Symposium (1996).

[Badger] L. Badger *et al.*, "A Domain and Type Enforcement UNIX Prototype", Proceedings of the USENIX 5th UNIX Security Symposium (1995).

[Bulba] Bulba and Kil3R, *Bypassing StackGuard and Stackshield*, Phrack Mag., V. 10, N. 56, 2000.

[Bouchareine] P. Bouchareine, *Format bugs*, Bugtraq mailing list: July 18 2000.

[Sekar] R. Sekar, T. Bowen and M. Segal, "On Preventing Intrusions by Process Behavior Monitoring", Proceedings of the USENIX Workshop on Intrusion Detection and Network Monitoring (ID '99).

Appendix

The check_rootproc function

```
int check_rootproc(struct inode *ino) {
   int cont=0,iproc=0,error=0;
   suidp_t * suidproc;
   efile_t f;
   suidp_t p;

    if ((IS_SETUID_TO_ROOT(current))||(IS_A_ROOT_DAEMON(current)))  {
      for (;cont<permitted.total;cont++) {
        if((permitted.lst[cont].efid.device==ino->i_dev)&&
           (permitted.lst[cont].efid.inode==ino->i_ino)) {
            if((permitted.lst[cont].efid.size==ino->i_size)&&
               (permitted.lst[cont].efid.modif==ino->i_mtime)) {
                suidproc=permitted.lst[cont].processes;
                for (iproc=1;iproc<=permitted.lst[cont].proc_nr;iproc++)  {
                     if (!strcmp(suidproc->suidp_id.comm,current->comm)) {
                        suidproc->suidp_id.count++;
                        return PSA;
                     }
                     if (iproc<permitted.lst[cont].proc_nr) {
                        suidproc=suidproc->next;
                    }
                }
            } else {
               error=EFNA;
               goto file_exe_unauthorized;
            }
        }
      }
      error=EXENA; /* EXE is not in the database */
      goto file_exe_unauthorized;
    }

    return PNS; /* the process is not setuid to root or root daemon */

    file_exe_unauthorized:
                   f.efid.device=ino->i_dev;
                   f.efid.inode=ino->i_ino;
                   f.efid.size=ino->i_size;
                   f.efid.modif=ino->i_mtime;
                   strncpy(p.suidp_id.comm,current->comm,
                           sizeof(p.suidp_id.comm));
                   p.suidp_id.count=1;
                   do {
                     while (writer_pid!=0){
                       cli(); /* interrupt disabled */
                       if (writer_pid!=0)
                         interruptible_sleep_on(&pid_queue);
                       sti();
                     }
                   } while (!atomic_access(&writer_pid,current->pid));
                   /* start of critical section */
                   do_setuid_put(&(f.efid),&(p.suidp_id),FAILURE);
                   writer_pid=0; /* end of critical section */
                   atomic_access(&writer_pid,0); /* release of the lock */
                   return error;
```

Domain and Type Enforcement for Linux

Serge E. Hallyn
College of William and Mary
hallyn@cs.wm.edu, http://www.cs.wm.edu/~hallyn
Phil Kearns
College of William and Mary
kearns@cs.wm.edu, http://www.cs.wm.edu/~kearns

Abstract

Access control in Linux is currently very limited. This paper details the implementation of Domain and Type Enforcement (DTE) in Linux, which gives the system administrator a significant advantage in securing his systems. We control access from domains to types, domain transitions, and signal access between domains, based on a policy which is read at boot time.

1 Introduction

Access control in Linux currently consists of traditional Unix permissions and POSIX capabilities[Caps-faq]. Domain and Type Enforcement (DTE) has been presented [DTE95, DTE96] as a useful method for enhancing access control. DTE groups processes into domains, and files into types, and restricts access from domains to types as well as from domains to other domains. Type access can be any of read, write, execute, create, and directory descend. Domain access refers the right to send signals as well as that to transition to a new domain. A process belongs to exactly one domain at any particular time. A process transitions to a new domain by executing a file which has been defined as an *entry point* to that domain. The three types of domain transitions are *auto*, *exec*, or none. If Domain A has auto access to domain B, and a process in domain A executes an entry point for domain B, then the process will be automatically switched to domain B. If domain A has exec access to domain B, then a process running under domain A can choose whether to switch to domain B on execution of one of B's entry points.

DTE can be considered an abbreviated form of classical capabilities[Dennis66]. In a system based upon classical capabilities, a process carries with itself a set of access rights to particular objects. At any point, a process can give up, or reclaim (if permitted) some of its capabilities. POSIX capabilities work similarly, but these capabilities are limited to a predefined subset of superuser access rights such as the ability to nice a process, boot the system or open a privileged (< 1024) port. In DTE, a process carries with itself only an indicator of the domain in which it runs, and this determines the process' access rights. A process can enter a new domain (and hence change its access rights) only upon file execution.

Trusted Information Systems has used DTE in its proprietary firewalls, but details of its implementation were not publicly available, and TIS appears to have stopped using DTE altogether. A group at SAIC has recently begun a DTE for Linux implementation[SAIC-DTE]. Jonathon Tidswell and John Potter[Tidswell97] submitted theoretical work on extending DTE to allow safe dynamic policy changes, but have attempted no implementation.

Presented here is our prototype implementation of DTE for Linux version 2.3.

2 Implementation

We have implemented a DTE prototype in the `2.3.28` Linux kernel. Our implementation of DTE attaches type information to VFS inodes and domain information to process descriptors (task structs). A DTE policy is read at boot time from the text file `/etc/dte.conf`.

Traditional UNIX permissions are still enforced. There are several reason for this, such as user and system administrator familiarity with traditional UNIX protection. Most importantly, however, DTE is designed to provide mandatory access control to protect a system from subverted superuser processes. A DTE policy to replace traditional UNIX access control would be very large and complex. However, one could completely void traditional access control by simply giving all users full access to all files. Similarly, one can bypass DTE by creating a DTE policy with only one type and one domain, and full access from the sole domain to the sole type.

2.1 Data Management

At boot time, we build a structure for each domain as specified in the DTE policy file. This structure contains information regarding permitted access to types, permitted transitions and signal access to other domains, and entry points. Every process' task structure will contain a pointer to the structure for the domain to which it currently belongs.

At this time we also create an array containing the names of all types. Types are then compared by the offset of the type name in this array. Every inode contains three pointers which either are NULL or point into this array. The three pointers represent the *etype*, *rtype* and *utype* values. The *etype* value is the type of this particular file or directory. The *rtype* represents the value of this directory and its children, whereas the *utype* represents only the type of its children.

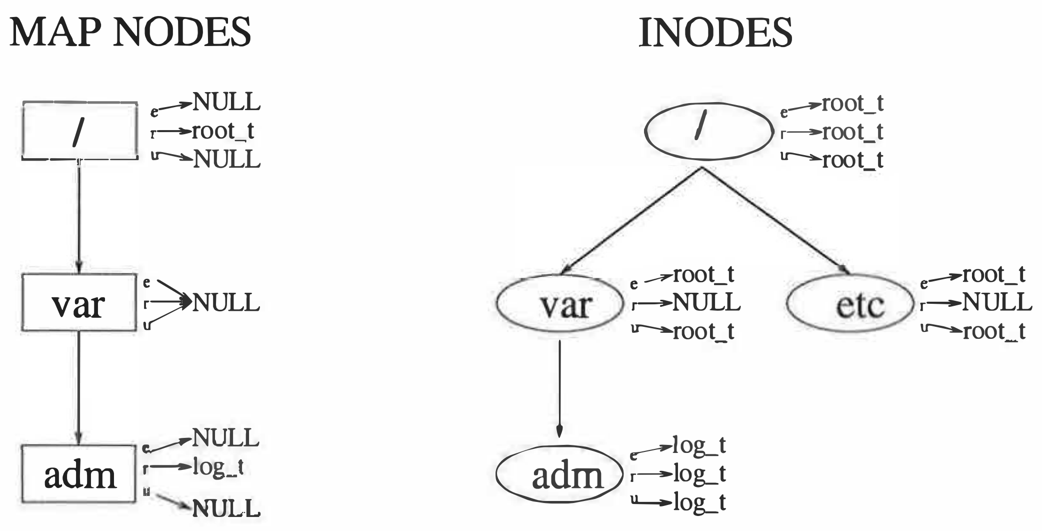
The type of a file is determined in one of three ways. First, if we have previously determined the type, then the inode's *etype* will be set and we simply use it. If this is the first time we are looking up this file, and a rule exists assigning it a type, then that rule is used. Finally, if a rule does not exist assigning a type to the file, then the values are inherited from the parent's *utype*[1]. The *etype* of an inode is always set. If there is no rule specifying the *etype* for the file, then the *etype* is set to the parent's *utype*. The *utype* of an inode must therefore also exist. It is set, in order of preference, to the assigned *utype*, the assigned *rtype*, or the parent's *utype*. The *rtype* of an inode is set only if a rule assigns an *rtype* to the inode's path.

Since type information always comes from either the DTE policy or from an inode's ancestors in the filesystem tree, no information needs to be added to the filesystem on disk. The type assignment rules are represented in memory by a tree of *map nodes* which is constructed at boot time from the type assignment rules in the policy. A sample tree for a particular set of rules is shown in Figure 1, along with the type information in corresponding inodes. The map nodes are only used to determine whether a rule exists binding a path to a type. Once an inode's type is set, subsequent lookups will not cause us to check the map nodes again.

2.2 Type Access Enforcement

When a process performs an *open* system call, the modified kernel checks for DTE permission before checking the standard UNIX permissions. We use the domain structure pointed to by the current task structure to check whether the current domain has the requested access to the type to which the file being opened belongs. If this access is granted, then we proceed to perform the normal UNIX checks. A check for DTE execute permission is delayed until the actual call to *execve*. If the execution causes an allowed domain transition, then this transition should occur before the check for execute access, since the new domain may be the only one allowed to execute the entry point.

[1]The type of the root of the filesystem is set explicitly in the DTE policy.



```
# extract from the sample policy:
default_rtype   root_t
assign  -r  /var/adm   log_t
```

Figure 1: Sample DTE assign rules and corresponding map nodes

Domain to type access information is kept in a set of hash tables. Each domain structure has a hash table keyed by the type name, and each entry lists the domain's access rights to the particular type. A type access check, therefore, consists of simply calculating the hash value of the type name to find the appropriate domain to type access entry for the current domain, and comparing the requested access to the permitted access. This is done regardless of username, so that the superuser is not exempt from the DTE policy.

2.3 Domain Access Enforcement

Our DTE implementation enforces restrictions on signals between processes in different domains. Each domain structure contains a linked list of *dte_signal_access* structures, which contain the signal number and a pointer to the domain to which it may be sent. One of these structures exists for every signal which may be sent to another domain. However, for the sake of abbreviation, setting the domain to null allows the specified signal to be sent to all domains, and setting the signal to 0 allows all signals to be sent to the specified domain.

2.4 Domain Transition Enforcement

Three types of domain transition are possible each time a file is executed. The first is an *auto*, or mandatory, domain transition. Since this must be automatic, it means that each time a process calls `execve`, we must check whether the file being executed is an entry point into a domain to which the current domain has *auto* access. The second type of transition is an *exec*, or user-requested, transition. This is facilitated by a new system call, `sys_dte_execve`, which takes an additional argument over `execve` containing the name of the requested domain. The third and default type of transition is the NULL transition, wherein the domain is not changed.

Domain transition information is kept in two types of structures, both linked from the domain structure. Since *auto* transitions must be checked for on every execve system call, the search for a particular pathname must be very quick. Therefore, each domain structure contains a hash table of the pathnames whose execution lead to *auto* domain transitions, along with the domain to be switched to.

The domain structure also has a linked list of structures representing allowed *exec* transitions. Since a dte_exec is a relatively rare, and user-requested, event, efficiency is not so critical, and we can elect for a more memory-efficient representation. Therefore we do not keep a hash table of every file which may cause an *exec* transition, but simply point to the domains to which a voluntary (*exec*) transition is allowed. To check for *exec* access to a domain, we must first check for an *exec* entry for the desired domain, then check whether the file being executed is an entry point for that domain.

3 Administration

Administering DTE consists of editing the policy, which is defined in the file /etc/dte.conf. The system must be rebooted to effect the changes[2].

The DTE policy file consists of several sections. We first enumerate the types and domains. Next we specify the default type for the filesystem root ("/") and its children, and the domain in which to run the first process (init). Following is the detailed definition of all domains. For each domain we specify the entry points, permitted type access, permitted domain transitions and permitted signals to processes in other domains. Finally we list the type assignment rules.

[2]Allowing the safe run-time changing of access control rules is a topic of some ongoing research[Tidswell97]

A sample policy file is in Figure 2. First we specify that there will be two types, *root_t* and *log_t*, and two domains, *common_d* and *log_d*. We set the default root rtype, hence the default type for the entire filesystem, to *root_t*. Next we set the type of the first process to *common_d*. We specify that the *log_d* domain will have one entry point, /sbin/syslogd, and should have read, execute and directory descend access to files of type *root_t* and read, write, execute, create and directory descend access to files of type *log_t*. For the domain *common_d*, we specify read, write, execute, create and directory descend access to files of type *root_t*, but only read access to files of type *log_t*. This domain also receives *auto* transition access to domain *log_t*, meaning that, on execution of /sbin/syslogd, a process in domain *common_d* will automatically be switched to domain *log_d*. Finally, the last statement assigns the type *log_t* to the directory /var/adm/log and all files thereunder.

4 The DTE API

Three additional system calls are provided to allow software to interact with DTE. The sys_dte_exec call was discussed earlier. A user may invoke sys_dte_gettype to learn the type associated with a file. Similarly, sys_dte_getdomain may be called to learn the domain associated with a process.

5 Performance

We measured the performance of both a DTE-enabled and a DTE-free 2.3.28 kernel for the execve and lookup_dentry system calls, the overhead imposed by the DTE-specific sys_dte_exec and dte_auto_switch system calls, and a full kernel compile. The following tests were run on a 400Mhz Pentium II (397.31 bogomips) with 512K L2 cache and 384M ram. Each test was run on a kernel compiled without DTE and one with DTE using the simple policy shown in Figure 3. We used the Pentium cycle clock for timing. All confidence

```
# this is a comment
types root_t log_t      # enumerate the types
domains common_d log_d  # enumerate the domains

default_rtype root_t    # default type for /
default_domain common_d # domain for process 0

# A domain is specified in n parts:
# spec_domain <domain_name> (entry points) (type access) (domain access) \\
#                  (signal access)
spec_domain log_d (/sbin/syslogd) (rdx->root_t rwxcd->log_t) () ()
#             ^      ^                 ^                          ^
#           (name) (entry point)    (type access)             signal access
spec_domain common_d () (rwxcd->root_t r->log_t) (auto->log_d) ()
#                                                      ^
#                                                domain access

assign -r /var/adm/log log_t   # assign type log_t to /var/adm/log
                               # and all files there-under
```

Figure 2: A sample DTE configuration file

intervals are 95%.

5.1 Permission

The `fs/namei.c:permission` kernel function is used before any file operations to check whether the user is authorized to perform the requested action. The code to check for domain to type access rights is located at the top of this function, so that DTE permissions are checked before standard UNIX permissions. As mentioned above, each domain has a hash table, keyed by type name, listing the domain's access rights to types. The DTE permission check is therefore very quick and constant time with respect to the number of types. Of course, it is linear with respect to the length of the pathname, as we need to first find the pathname and then hash it.

The first time it is called on a particular file or directory, however, the *etype* may not yet have been set. In this case, we must check for a type assignment rule or, if such a rule does not exist, set the type from the parent directory. Furthermore, if the parent directory does not exist then we must first do the same for it, and so on until a directory is associated with a type assignment rules or has its type set.[3]. The DTE type assignment rules are kept in a tree format analogous to the filesystem tree, as shown in Figure 1. The children are currently not sorted, so that a large number of assignment rules for files under a single parent directory could impact performance. However, for normal cases this lookup should be reasonably quick.

We timed the upper part of the kernel function `fs/namei.c:permission`, where the DTE code is located. Over the course of a boot sequence, several repeats of the lookup test above, some general milling around, and a shutdown, the DTE code added 1578 ± 400 clock cycles to each `permission` call.

5.2 lookup_dentry

The time required to look up a given pathname greatly affects the subjective performance of the system. The function `fs/namei.c:lookup_dentry`, which performs this task in the Linux kernel, is affected by DTE in two places. First, for each subdirectory in a pathname, `lookup_dentry` calls per-

[3]As must eventually be true since the filesystem root has a defined type

```
types root_t login_t user_t test_t spool_t tripwire_t
domains root_d login_d user_d test_d tripwire_d
default_d root_d
default_et root_t
default_ut root_t
default_rt root_t
spec_domain root_d (/bin/bash /sbin/init /bin/su) (rwxcd->root_t rwxcd-
>spool_t \
      rwcdx->user_t) (auto->login_d auto->tripwire_d)
spec_domain login_d (/bin/login /bin/login.dte) (rxd->root_t rwxcd-
>spool_t) \
      (exec->root_d exec->user_d exec->test_d)
spec_domain user_d (/bin/bash /bin/tcsh) (rwxcd->user_t rwxd->root_t rwxcd-
>spool_t) \
      (exec->root_d exec->test_d)
spec_domain test_d (/bin/bash) (rwxcd->test_t rdx->user_t rwdx-
>root_t rwxcd->spool_t) \
      ()
spec_domain tripwire_d (/bin/tripwire) (rwxcd->tripwire_t rxd->user_t rxd-
>spool_t \
      rxd->root_t) ()

assign -r /etc/tripwire tripwire_t
assign -r /var/spool/tripwire tripwire_t
assign -u /home user_t
assign -u /tmp spool_t
assign -u /var spool_t
assign -u /dev spool_t
assign -u /scratch user_t
assign -r /dte_test_dir test_t
# next one is a test - user_d should *not* see it since no 'd' to /dte_test_dir
assign -e /dte_test_dir/aha user_t
```

Figure 3: DTE policy used for performance tests

`mission` to check for execute access. Second, if the types for the deepest path element being looked up have not been set, then we must set them, using the same function we use above in `permission`.

We timed `lookup_dentry` on a set of pathnames ranging in depth from 1 to 9 components, both for fully existing and fully nonexistent pathnames. For the first execution, each component of each pathname was uncached. On subsequent executions, all path components and their corresponding DTE type information were (naturally) cached.

The results can be seen in figures 4, 5, 6 and 7. For the case of a lookup for uncached filenames, results appear to be rather unpredictable. If this appears to be more true for existing file lookup than for nonexistent files, this is a result we should have predicted by our method of testing. We tested the lookup for each set of pathnames, then rebooted, and repeated the test, eleven times in all. However, for the nonexistent filename lookup, a part of the pathname was legitimate. This piece was looked up uncached only for our first test after reboot, which was for the first table entry in figure 6. Since the DTE kernel was faster than the plain kernel more often than it was slower, it appears safe to say that disk i/o completely overshadows any time spent setting DTE types from map rules and parents.

For cached lookups, the DTE kernel appears to do slightly better than twice as long as the plain kernel.

5.3 *auto* Domain Transitions

Upon file execution, we must check whether the requested execution should cause a mandatory domain switch. This is done using `kernel/dte.c:dte_auto_switch`. As previously mentioned, this function must be fast as it is called with every file execution. Therefore, it simply hashes the name of the executable to check for an entry in a table of gateways, or executables which cause an automatic domain switch.

We compiled a kernel which timed the execution of `dte_auto_switch`. If a particular domain has no gateways, then `dte_auto_switch` does not bother to hash the typename, so that the `dte_auto_switch` call during `execve` takes 308 ± 6 clock cycles. If there are gateways, then we must search the hash tables. While we tested using domains with a variable number of permitted auto switches,[4] this number does not affect the running time of `dte_auto_switch`, which is 6655 ± 166 clock cycles. Since this function does not exist in the plain Linux kernel, its running time must be considered pure overhead to file execution.

5.4 *exec* Domain Transitions

The least efficient of all the code added with DTE certainly sits in `kernel/dte.c:sys_dte_exec`. First, the user provides the name of a domain to switch to. Since domains are currently not kept hashed or in any order, the lookup for the corresponding domain structure is $O(d \times m)$, where d is the number of defined domains and m is the maximum length of any domain name. Next, we search another unsorted list, containing the domains to which the current domain may voluntarily switch, to check whether the domain switch is legal. Then we search a third list, containing valid entry points for the destination domain.

In order to measure the amount of time required to check for an *exec* domain switch, we set up 12 domains, with entry points numbering $2, 4, 6, 8, 10, 12, 14, 16, 18, 20$ and 30, where two domains had 30 entry points. Then we performed an *exec* domain switch into each of these domains, and measured the time between the start of `sys_dte_exec` and its call to `sys_execve`. Since entry points are stored unsorted, the 12th domain's list of entry points contained the entry point which we actually executed last, whereas all others listed it first. The results for 10 trials (excluding the first of eleven since the entry

[4] Mainly to test for a bad implementation of poor hash function.

# Path Elements	plain	DTE
1	14239 ± 115	3715106 ± 3764708
2	12982865 ± 740602	9443087 ± 3764708
3	12328686 ± 3305948	17481580 ± 18583745
4	16894558 ± 1323098	15491659 ± 3111030
5	11514025 ± 1124946	11145553 ± 2406925
6	14939328 ± 837802	13684836 ± 2800215
7	21312794 ± 17210467	12670629 ± 3303167
8	6362366 ± 2955509	4912447 ± 2608638
9	33647759 ± 20269841	19853653 ± 4192949

Figure 4: existing file lookup, first runs

# Path Elements	plain	DTE	% increase
1	4450 ± 56	8222 ± 71	85
2	5085 ± 67	8925 ± 81	76
3	5223 ± 75	10798 ± 105	107
4	6475 ± 102	12076 ± 151	87
5	7097 ± 101	13463 ± 124	90
6	7747 ± 126	14605 ± 171	89
7	8374 ± 155	15918 ± 167	90
8	9194 ± 99	17415 ± 144	89
9	9867 ± 254	18602 ± 186	89

Figure 5: existing file lookup, cached runs

# Path Elements	plain	DTE
1	11437289 ± 911243	7667678 ± 3621479
2	10045 ± 1270	12319 ± 239
3	9421 ± 345	12260 ± 322
4	9343 ± 378	12482 ± 281
5	9911 ± 1171	35990 ± 43904
6	9934 ± 1298	12036 ± 371
7	9299 ± 287	12789 ± 1147
8	9956 ± 1151	12929 ± 1231
9	9236 ± 275	12336 ± 543

Figure 6: non-existent file lookup, first runs

# Path Elements	plain	DTE	% increase
1	4508 ± 90	8221 ± 57	82
2	4414 ± 136	8270 ± 120	87
3	4334 ± 120	8240 ± 103	90
4	4330 ± 90	8270 ± 133	91
5	4267 ± 177	8236 ± 123	93
6	4247 ± 184	8231 ± 133	94
7	4326 ± 63	8206 ± 137	90
8	4416 ± 79	8267 ± 99	87
9	4415 ± 106	8240 ± 154	87

Figure 7: non-existent file lookup, cached runs

point needed to be read from disk) are shown in Figure 8.

The first 11 domains each executed the first file in the respective domains' linked list of entry points. The difference in performance is due to the analogous problem with the list of allowed *exec* transitions.

For domains with what we believe to be a realistic number of entry points (1-8), `sys_dte_exec` takes about 4 times as long as `dte_auto_switch`. Clearly, performance will be greatly improved when we store entry points, domains, and allowed *exec* transitions in a data structure which allows quicker lookups. This will be a simple but low priority improvement, since a policy must be quite large for the effects to become noticeable, and a `sys_dte_exec` call is a rare event.

5.5 execve

To time the kernel function `fs/exec.c:do_execve`, we wrappered it and took a timestamp before and after the real function call. In this way we measure the full time for file execution including such details as the time to load library files. For more fine-grained measurements of specific parts of this process, we later measure the time to check for the *auto* domain switch, a user-requested domain switch and filename lookup.

The command

```
/bin/echo -n .
```

was executed 500 times. Execution time for the first run was an order of magnitude larger than for subsequent runs, both with and without DTE. This is to be expected since some library files as well as executable `/bin/echo` may not yet have been loaded from disk. The same thing occurs for later performance tests. Since this is independent of the DTE code and serves only to hide the performance impact of DTE, we will, in all subsequent tests, ignore the first execution after boot.

The DTE code introduces a 10% overhead. The table in Figure 9 shows the timing results.

5.6 *make bzImage*

Finally we turned off all micro-performance measurements and used `/usr/bin/time` to determine the performance on a kernel make on both DTE and non-DTE enabled kernels. The plain `2.3.28` kernel took 5 minutes and 55 seconds for the first compile, and 5:35 ± 0.384387 for 14 subsequent compiles, while the DTE-enabled kernel required 5 minutes and 56 seconds for the first compile and 5:36 ± 0.205464 for subsequent compiles.

Clearly a new access control system cannot be added without affecting performance. The above sections, in testing specifically the areas of the kernel where code was added, might make the performance impact of DTE seem more significant than it really is. This result shows that, when amortized over the course of a realistic activity, which includes heavy file opening, creation and execution as well as heavy computation and file i/o, the amount of overhead, one second for every six minutes, is negligible.

6 Real Attacks

To show the effectiveness of our DTE implementation, we picked a recent, high-profile vulnerability, the buffer overflow in *wu-ftpd*[CERT-ftpd], and showed how our implementation of DTE can prevent an attacker from obtaining a root shell. Our goal was to show that we could protect the system from the *wu-ftpd* vulnerability (the posted exploits as well as future or hand-crafted ones) without modifying the binary. In order for ftp to retain its full functionality, it would need to be made DTE-aware so that it could, like login, allow ftp

# entry points	Clock cycles for `sys_dte_exec`
2	22692 ± 184
4	22777 ± 133
6	23186 ± 269
8	23432 ± 256
10	24099 ± 336
12	23946 ± 123
14	24367 ± 238
16	24503 ± 120
18	24841 ± 125
20	24978 ± 133
30	25282 ± 142
30 (entry point last)	32427 ± 259

Figure 8: Dependence of `sys_dte_exec` performance on number of entry points.

	Non-DTE	DTE
First Execution	4221290 ± 911041	4545504 ± 730168
Subsequent executions	195591 ± 3919	215549 ± 4624

Figure 9: Time in clock cycles to run *echo*.

to transition into the domain associated with a user being authenticated[5]. We did not do this, but set protections such that users can retrieve files from, if not deposit files onto, the server. Anonymous ftp is fully functional.

The policy shown in Figure 10 prevents domain *ftpd_d* from executing any system binaries other than `/usr/sbin/in.ftpd` and binaries located under `~ftp/bin/` (lines 19-21). These files are defined to be of the type `ftpd_xt` (lines 29 and 30), which the domain *ftpd_d* may execute but not write (line 20). Only *ftpd_d* may execute this type (lines 9-21), and *root_d* automatically switches to *ftpd_d* on execution of `/usr/sbin/in.ftpd` (line 12), since that is an entry point to *ftpd_d* (line 19). The exploits to be found on the internet to take advantage of this vulnerability will therefore fail, as they expect to be allowed to run `/bin/sh`. Nor can a script be written to upload and run a Trojan horse, since the only types which *ftpd_d* is allowed to write may not be executed by anyone.

The script which we tested was `wuftpd2600`, which can be found at `http://www.securityfocus.com`. It connected to our test machine, and exploited the buffer overflow. However, the DTE-enabled kernel refused to allow the *ftpd_d* domain to execute `/bin/sh`. The script therefore hung, and the system was not compromised. The error messages in Figure 11 were sent to *syslog*. In contrast, the plain `2.3.28` kernel happily provided a root shell.

[5]Of course, to do this on a system which we are attempting to make secure, we would begin by using a version of ftp which does not send plaintext passwords.

7 Status and Future Work

Our implementation of DTE for Linux is functional. It reads a policy file at boot time and enforces domain to type access as well as domain transitions. We have not implemented DTE for networking.

7.1 Administration

First, we must extend our policy parser to allow easier and abbreviated entry of more

```
01 # ftpd protection policy
02 types root_t login_t user_t spool_t binary_t lib_t passwd_t shadow_t dev_t \
03       config_t ftpd_t ftpd_xt w_t
04 domains root_d login_d user_d ftpd_d
05 default_d root_d
06 default_et root_t
07 default_ut root_t
08 default_rt root_t
09 spec_domain root_d (/bin/bash /sbin/init /bin/su) (rwxcd->root_t rwxcd-
>spool_t \
10       rwcdx->user_t rwdc->ftpd_t rxd->lib_t rxd->binary_t rwxcd-
>passwd_t \
11       rxwcd->shadow_t rwxcd->dev_t rwxcd->config_t rwxcd->w_t) (auto-
>login_d \
12       auto->ftpd_d) (0->0)
13 spec_domain login_d (/bin/login /bin/login.dte) (rxd->root_t rwxcd-
>spool_t \
14       rxd->lib_t rxd->binary_t rwxcd->passwd_t rxwcd->shadow_t rwxcd-
>dev_t \
15       rxwd->config_t rwxcd->w_t) (exec->root_d exec->user_d) (14->0 17-
>0)
16 spec_domain user_d (/bin/bash /bin/tcsh) (rwxcd->user_t rwxd->root_t \
17       rwxcd->spool_t rxd->lib_t rxd->binary_t rwxcd->passwd_t rxwcd-
>shadow_t \
18       rwxcd->dev_t rxd->config_t rwxcd->w_t) (exec->root_d) (14->0 17-
>0)
19 spec_domain ftpd_d (/usr/sbin/in.ftpd) (rwcd->ftpd_t rd->user_t rd-
>root_t \
20       rxd->lib_t r->passwd_t r->shadow_t rwcd->dev_t rd->config_t rdx-
>ftpd_xt \
21       rwcd->w_t d->spool_t) () (14->root_d 17->root_d)
22 assign -u /home user_t
23 assign -u /tmp spool_t
24 assign -u /var spool_t
25 assign -u /dev dev_t
26 assign -u /scratch user_t
27 assign -r /usr/src/linux user_t
28 assign -u /usr/sbin binary_t
29 assign -e /usr/sbin/in.ftpd ftpd_xt
30 assign -r /home/ftp/bin ftpd_xt
31 assign -e /var/run/ftp.pids-all ftpd_t
32 assign -r /home/ftp ftpd_t
33 assign -e /var/log/xferlog ftpd_t
34 assign -r /lib lib_t
35 assign -e /etc/passwd passwd_t
36 assign -e /etc/shadow shadow_t
37 assign -e /var/log/wtmp w_t
38 assign -e /var/run/utmp w_t
39 assign -u /etc config_t
```

Figure 10: A DTE policy to protect from *wu-ftpd*, with line numbers added.

```
Aug  4 13:12:03 wicked kernel: do_exec: d_t_check_x re-
turned 1(exec denied).
Aug  4 13:12:03 wicked kernel: do_exec: domain ftpd_d type root_t.
```

Figure 11: Error messages resulting from attempted *wu-ftpd* exploit.

complicated policies. Next, we plan to create tools to help a system administrator graphically create and view DTE policies and detect possible security risks. Examples of such risks might include a domain which is permitted to enter another domain as well as write one of the other domain's entry points, or a domain which has *auto* access to two domains which share an entry point.

7.2 Rename

Badger et al.[DTE95] suggest dynamically changing the DTE policy as certain events occur. For example, a particular file (`/var/adm/topsecretlog`) might be tightly protected by a particular type. If this file is then moved to `/tmp`, then Badger et al. suggest that a rule should be added to keep the file under its original type. Alternatively, they suggest that renames across type boundaries could be forbidden.

We currently go the lazy route. If a domain has permission to two types, and a process running in that domain chooses to move a file from a directory belonging to one type to that belonging to another, then the file's type simply changes. Since the person (or process) moving the file had the permission to do so, we trust it to understand the implications.

The more dangerous problem lies with hard links. Since hard links provide no notion of one name being superior to another, the type of an inode with multiple corresponding filenames is currently determined based upon the name first looked up.[6] We can prevent creation of hard links across type boundaries, however a change in policy can thwart this defense quite easily. We will, in future versions, allow the system to add its own type assignment rules (which will be necessary for several other desirable features), and plan to use this capability to implement a better resolution of the problem with hard links.

[6]Note there is no analogous problem with soft links, since these do provide a notion of a single correct pathname.

7.3 Filesystem-Defined Policies

When a partition is mounted into the filesystem tree, it fits into the tree defined by type rules depending on where it is mounted. For example, mounting a partition under `/tmp` instead of `/mnt` might completely change access permissions to the partition. It seems helpful to allow the filesystem on a partition to specify certain type assignment rules which apply only to the partition. Badger et al[DTE96] also added a DTE configuration section allowing a DTE administrator to limit mount points for partitions, which should be trivial for us to add as well.

8 Availability

DTE for Linux is freely available as a patch to `2.3.28` at `http://www.cs.wm.edu/~hallyn/dte`.

References

[AC-sum] R. Sandhu, *Access Control: The Neglected Frontier*, First Australasian Conference on Information Security and Privacy, 1996.

[Caps-faq] Alexander Kjeldaas, *Linux Capability FAQ v0.1*, `http://www.uwsg.indiana.edu/hypermail/linux/kernel/9808.1/0178.html`, (1998).

[CERT-ftpd] *Cert Advisory CA-2000-13: Two Input Validation Problems in FTPD*, `http://www.cert.org/advisories/CA-2000-13.html`.

[Dennis66] Jack B. Dennis and Earl C. Van Horn, *Programming Semantics for Multiprogrammed Computations*, Communications of the ACM, March 1966, pp. 143-155.

[DTE95] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker and Sheila

A. Haghighat, *A Domain and Type Enforcement UNIX Prototype*, Fifth USENIX UNIX Security Symposium Proceedings, Salt Lake City, Utah, June 1995.

[DTE96] Kenneth M. Walker, Daniel F. Sterne, M. Lee Badger, Michael J. Petkac, David L Shermann, Karen A. Oostendorp, *Confining Root Programs with Domain and Type Enforcement(DTE)*, Sixth USENIX UNIX Security Symposium, 1996.

[SAIC-DTE] *Domain and Type Enforcement*, `http://research-cistw.saic.com/cace/dte.html/`.

[Tidswell97] Jonathon Tidswell and John Potter, *An Approach to Dynamic Domain and Type Enforcement*, 2nd Australasian Conference on Information Security and Privacy, 1997.

Piranha Audit: A Kernel Enhancements And Utilities To Improve Audit/Logging

Vincenzo Cutello, Emilio Mastriani, Francesco Pappalardo
Department of Mathematics and Computer Science
University of Catania, Catania, Italy
{cutello, mastriani, francesco}@dmi.unict.it

Abstract

This paper presents a mechanism to enrich logging as required in TCSEC [1] document to detect and stop possible intrusions based on typical attacks and to protect the sensible audit data from deletion/modification even in root compromise situation.

After installing Piranha Audit, administrators will have a solid infrastructure for improving security and resistance to penetration, with only modest performance penalties.

We present experimental results of the advantages of this solution and the performance impact of the mechanism.

1 Introduction

The main purpose of this work is to present a systematic solutions to the persistent problems of securing and improving the Audit and Logging capabilities.

Moreover, we will present a collection of suites to perform Intruder Detection and a proposal to protect the system against Buffer Overflow Attacks. Covert Storage Channel Analisys is currently under study.

The basic problem is that, in a root compromise case, all audit data can be deleted or altered, trashing the collected informations even if they respect the TCSEC requirements.

An important question is: does anybody have the time to inspect hundreds of lines generated by Audit/Logging system? We provide a collection of utilities that analyze in real time such data and take the least disruptive action to terminate the event that may corrupt the system integrity.

In doing so we will try to meet Dision B, Class 3 TCSEC requirements.

Section 2 decribes the state of the art in Linux about Audit and Logging, the typical attacks against the integrity and security of the system and what are the TCSEC requirements in detail.

Section 3 shows how Piranha Audit helps system administrators to detect what has happened and how Intruder Detection System defends against some more dangerous attacks. Kernel patches applied and a quick description of the suite of user utilities will be also provided.

Section 4 presents performance and penetration testing. Section 5 describes related works. Finally section 6 presents our conclusions.

2 General overview

The standard Linux Kernel meets Division C, Class 2 "partially" in Audit context, since there is no system routine which records events of object introduction or deletion.

Once this problem was solved, to reach Division B, Class 1:

(a) the audit record will have to include, for each event that either introduces an object into a user's address space ot it deletes an object, the name of the object and the object's security level.

(b) Moreover, the system manager would have to be able to selectively audit the actions of any one or more users based on individual identity and/or object security level.

(c) Finally, it must be possible to audit any override of human-readable output markings.

To reach Class 3,

(d) one of the required features is the presence of a mechanism that is able to monitor the occurrence or accumulation of security auditable events that may indicate an imminent violation of security policy. This mechanism will have to be able to immediately notify the security administrator when thresholds are exceeded, and, if the occurrence or accumulation of these security relevant events continues, the system will have to take the least disruptive action to terminate the event.

(e) Moreover, we would need some mechanisms for the identification of events that may be used in the exploitation of the usage of covert storage channels.

In this paper, we will describe an extension of the standard Linux Kernel to reach Division C, Class 2 and that solves problems (a)-(d) as well. Problem (e) currently is solved for a particular case: File Flag Communication[1], but this needs more work.

Now we will describe a list of typical attacks [2].

- A system cracker telnets to the next site on his hit list. "guest – guest", "root – root", and "system – manager" all fail. It does not matter. A lot of sites have easy passwords to crack, based on user name, birth date and so on.
- NFS-Attacks. For instance, running showmount on a target reveals that /export/foo is exported to the world. In this case you can put an .rhosts entry in the remote guest home directory, which will allow you to login to the target machine without having to supply a password!
- Anonymous ftp attacks. Vulnerabilities in ftp are often a matter of incorrect ownership or permissions of key files or directory.
- X windows attacks. If not protected properly (i. e. via xhost or magic cookie mechanisms) window displays can be captured or watched.
- DoS[2] attacks. These type of attacks do not involve a penetration in a system. They slow or block a net service or the entire system.

[1]With this term we intend a illegal communication from root to user processes based on file presence that indicates, for example, a bit information.

[2]Denial of Service

- Sendmail attacks. Sendmail is a very complex program that has a long history of security problems, i. e. running the "decode" alias is a security risk: it allows potential attackers to overwrite any file that is writable by the owner of that alias, often daemon, but potentially any user.
- "hosts.equiv" attacks. The hosts recorded in this file are trusted: for example if a login request come from a site recorded in hosts.equiv file, there is no need to supply a password. Any form of trust can be spoofed.
- Buffer exploit attacks. If a malicious user finds a buffer overflow in a suid utility, he can gain root privilege.
- Password sniffing. The telnet sessions do not use any form of encryption; so an attacker can sniff the password during a telnet session.

New forms of attacks appear every day. This list can only be a short example.

3 Piranha Audit details

Why would you want to meet the TCSEC requirements? An Audit/Logging file that respects TCSEC layout provides detailed informations as described above. Moreover Piranha Audit protects sensible data against deletion/modification at root level and phisycal disk management (fdisk, format, kernel image replacement, boot the system from floppy). To allow these operations and dumping the Piranha_Audit.log, it's needed Pirahna Manager operator.

He/she is a trusted person that knows the Piranha password that is needed to complete Piranha Audit management sessions.

Only he/she can change the Piranha password. We emphasize that just the root or just Piranha Manager cannot assolve these rules: the execution of any Piranha management session (needs root privileges) requires the Piranha password.

Table 1 shows the files used and kernel protected by Piranha Audit.

This high level of protection has been obtained by applying patches to 2.2.14 Linux Kernel shown in table 2, where PM stands for Piranha Manager and SU for Super User.

Table 1: Piranha Audit files.

Files	Description
Piranha_Audit.log	Contains all sensible data from Audit/Logging System
syslog.conf	Configuration file for syslogd daemon
Piranha_FSCF_DB.md5	Collects MD5-fingerprint for critical file system objects
Piranha_SETUID-GID.db	Maps all SETUID-GID root files
Piranha_MD5_Digest_Creator	Utility that uses MD5 algorithm to create digital sign
Piranha_System_Shutdown	Utility to shutdown the machine in critical events
Piranha_Password	Contains the password for Piranha Manager operator

Table 2: Protection modes.

Protected Files	Patched Files	User Level	SU Level	SU+PM Level
Piranha_Audit.log	namei.c, open.c	—	r–	rd-
syslog.conf	namei.c, open.c	—	r–	rw-
Piranha_FSCF_DB.md5	name.c, open.c	—	r–	rw-
Piranha_SETUID-GID.db	namei.c, open.c	—	r–	rw-
Piranha_MD5_Digest_Creator	namei.c, open.c	—	r–	rx-
Piranha_System_Shutdown	namei.c, open.c	—	r–	rx-
Piranha_Password	namei.c, open.c	—	r–	rx-

r=read
d=dumping
x=execute

In "namei.c" and "open.c" we have also introduced a C routine that allows syslogd daemon to open Piranha_Audit.log in append only mode. The TCSEC layout is kept byinserting "printk" calls in "namei.c", "open.c", "pipe.c" in correct locations.

The "exec.c" has been patched to detect possible buffer exploit attacks. Suppose that a malicious user has exploited a setuid program. He/she produces "a.out" program that uses this bug to obtain root access. The program does its work and executes a root shell. Piranha Audit detects a particular situation: UID –> 500, GID –> 100, ***EUID –> 0,*** EGID –> 100. There is an anomaly: an inconsistence between UID and EUID; a kernel trap is executed. The user session will be terminated and the account will be locked.

The patched "signal.c" does not allow to kill the Piranha Guardian, detailed below in table 3 with a quick description of Intruder Detection Suite, where IDS stands for Intruder Detection System.

The Simple Watcher utility allows an automatic log analysis detecting patterns that implies an anomaly status.

When it is detected, Simple Watcher sends an Alert Message to Piranha Audit subsystem that takes the least disruptive action to terminate the event.

It is possible to configure rensponses to certain auditable events and to make the PM protection of key files configurable setting the Simple Watcher config file.

4 Performances and penetration testing

Examples 1, 2, 3, 4 and 5 show the behavior of Piranha Audit in some cases of intrusions attempts.

Example 1: Gaining a root shell.

- *Jul 9 10:07:32 SecureHost kernel: Piranha Audit: Warning: the object /bin/.su, executed by UID: 500, GID: 100 is set-uid!*
- *Jul 9 10:07:33 SecureHost su[3196]: + tty3 emilio-root*

Example 2: Attempt to remove Audit/Logging archive.

- *Jul 9 10:07:46 SecureHost kernel: Piranha Audit: Object delete command issued from UID 0, GID 0, object name: Piranha_Audit.log.*
- *Jul 9 10:07:46 SecureHost kernel: Piranha Audit: Owner of object is: UID 0, GID 0.*
- *Jul 9 10:07:46 SecureHost kernel: Piranha Audit: Object i-node mode is: 33188.*
- *Jul 9 10:07:46 SecureHost kernel: Piranha Audit: Unauthorized access (delete command) to security event file from UID: 0 GID: 0 detected.*

Example 3: Attempts to delete/link/overwrite Piranha_Audit.log.

- *Jul 9 10:08:59 SecureHost kernel: Piranha Audit: Read access of security event file detected from UID: 0 GID: 0.*
- *Jul 9 11:16:42 SecureHost kernel: Piranha Audit: Hard link not allowed for security event file from UID: 0 GID: 0.*
- *Jul 9 11:17:04 SecureHost kernel: Piranha Audit: Unauthorized or incorrect use of security event file detected from UID: 0 GID 0.*
- *Jul 9 11:18:14 SecureHost kernel: Piranha Audit: Unauthorized or incorrect use of security event file detected from UID: 0 GID: 0.*
- *Jul 9 11:18:36 SecureHost kernel: Piranha Audit: Unauthorized or incorrect use of security event file detected from UID: 0 GID 0.*
- *Jul 9 11:18:55 SecureHost kernel: Piranha Audit: Attempt to create confusion with special object (mknod system call) and protected Piranha Audit files detected from UID 0, GID 0.*

Example 4: Fdisk attempt.

- *Jul 9 11:23:45 SecureHost kernel: Piranha Audit: (ALERT LEVEL 3) Attempt of disk management without correct security procedure detected from UID: 0 GID: 0.*

Example 5: Satisfying TCSEC layout.

- *Jul 12 15:41:30 SecureHost kernel: Piranha Audit: Object introduction detected from UID 500, GID 100, object name is: trial.*
- *Jul 12 15:41:30 SecureHost kernel: Piranha Audit: Owner of object is: UID 500, GID 100.*

Table 3: IDS utilities.

Utility	Quick description
Piranha_Account_Locker	Locks an account after compromised events
Piranha_Intruder_Killer	Terminates work session of a buffer exploit compromised user
Piranha_MD5_Digest_Creator	Creates md5 finger-print
Piranha_PWD_Creator	Sets the Piranha Manager Password
Piranha_SETUID-GID_Checker	Controls every 60 minutes the root SETUID-GID map
Piranha_SETUID-GID_Init	Initializes root SETUID-GID database file
Simple Watcher [9]	Instructs Piranha about Alert Level reactions
Piranha_System_Shutdown	Halts the machine in critical situation
Piranha_Dumper	Allow under root+PM privileges file system management
Piranha_FSC	Protects critical files against modification/trojan horse attacks
Piranha_FSC_Init	Initializes the database with MD5 signs of critical files
Piranha_Guardian	Controls that all IDS works correctly. It cannot be killed
Piranha_Init	Script that coordinates the execution of IDS
Piranha_Overflow_Checker	Checks for dimension overflow of Piranha_Audit.log
Piranha_PG_PID_Search	Searches for suitable PID for Piranha_Guardian
Piranha_PID-UID_Finder	Gets from PID its owner (UID)

- *Jul 12 15:41:35 SecureHost kernel: Piranha Audit: Object delete command issued from UID 500, GID 100, object name: trial.*
- *Jul 12 15:41:35 SecureHost kernel: Piranha Audit: Owner of object is: UID 500, GID 100.*
- *Jul 12 15:41:35 SecureHost kernel: Piranha Audit: Object i-node mode is: 33188.*
- *Jul 12 15:42:42 SecureHost kernel: Piranha Audit: Special object introduction (mknod system call) detected from UID 500, GID 100, object name is: pipe.*
- *Jul 12 15:42:42 SecureHost kernel: Piranha Audit: Device number of object is: 0.*
- *Jul 12 15:42:42 SecureHost kernel: Piranha Audit: Object i-node mode is: 33261.*

Below we show the Piranha Audit System behavior to underline the performances under different conditions.

5 Related works

Anderson [3] first proposed using audit trails to monitor system activity. The use of existing audit records suggested the development of simple tools to check for unauthorized access to systems and files.

Bonyun [4] argued that a single, well-unified logging process was an essential component of computer security mechanisms.

Picciotto [5] presents a sophisticated audit capability for a Compartmented Mode Workstation.

Intrusion detection systems that focus on anomalous behavior have also driven research in auditing and logging. Axent Technologies [7] has presented IDS in Unix and NT platforms, but nothing for Linux.

Tripwire facility from the COAST [8] project at Purdue University can take care of the file system, but it can only report problems: it does not take any action to terminate the dangerous event.

6 Conclusions

We have presented Piranha_Audit, a systematic solution to the persistent problems of securing and improving the Audit and Logging capabilities, that prevents a broad class of buffer overflow security attacks from succeeding.

Its most important futures are that it denies the deletion/modification of protected files even in a root compromised situation; with TCSEC layout, the system administrator has a powerful method to investigate; intrusion detection is critical in today's complex enterprises. Attempting to manually review audit trails is hopelessly

Table 4: Main memory details (in bytes).

Total	Used	Free	Shared	Buffers	Cached
64716800	63213568	1503232	24977408	1347584	41750528

Table 5: CPU Info.

processor	0
vendor_id	GenuineIntel
cpu family	586
model	2
model name	Pentium MMX
stepping	3
cpu MHz	200.457340
fdiv_bug	no
hlt_bug	no
sep_bug	no
f00f_bug	yes
coma_bug	no
fpu	yes
fpu_exception	yes
cpuid level	1
wp	yes
flags	fpu vme de pse tsc msr mce cx8
bogomips	80.08

Table 6: Stress Testing.

CPU Stress Test	PASSED
Disk Stress Test	PASSED
CPU+Disk Stress Test	PASSED

CPU STATES: 62 processes: 48 sleeping, 14 running, 0 zombie, 0 stopped 98.2% user, 1.7% system, 0.0% nice, 0.0% idle

DISK USE: dd if=/dev/zero of=trial count=400000

PASSED means that Piranha Audit System behavior is correct how in no stress situation.

time-consuming and a losing battle given the number of systems and different types of audit trails. Today we need automated intrusion detection tools. Digitals finger print have produced with MD5 [6] algorithm, one of the best in its area.

All this with little performances degradation how is showed in the following figure.

7 Acknowledgements

We would like to thank the anonymous referees for their valuable comments.

Figure 1: Performances.

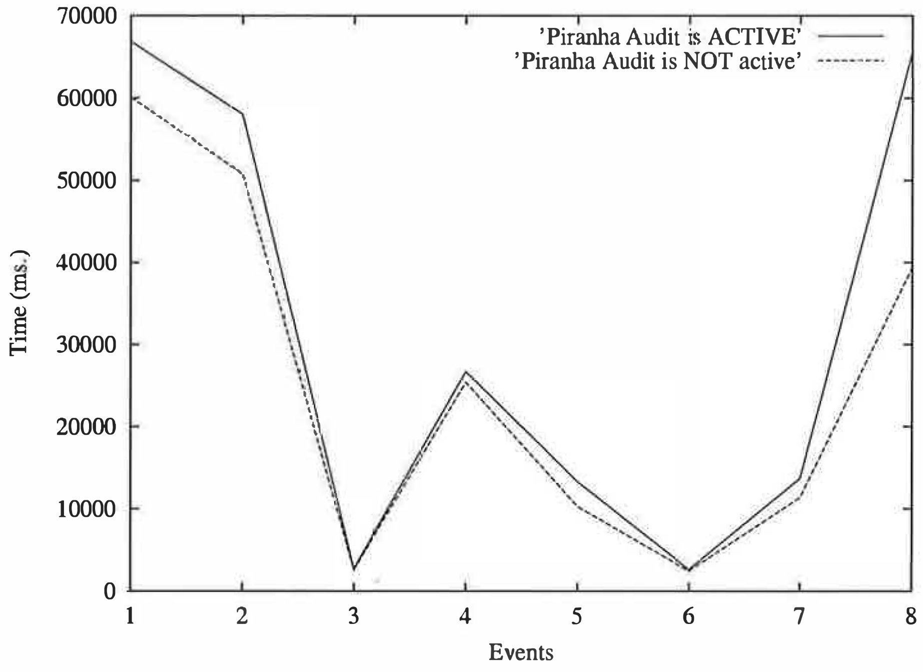


Table 7: Performance keywords

Event	Keywords
find | grep lyx	1
Pirannha Audit compile process	2
latex work.tex	3
Starting an X session	4
netscape	5
lyx	6
gimp	7
Linux Boot	8

References

[1] *Department of Defense Trusted Computer Systems Evalutation Criteria*. DoD 5200.28-STD, December 1985.

[2] D. Farmer and W. Venema, *Improving the Security of Your Site by Breaking Into It*, USENET posting, December 1993.

[3] James P. Anderson, *Computer Security Threat Monitoring and Surveillance,* James P. Anderson Co. , Fort Washington, PA (1980).

[4] David Bonyun, *The Role of a Well-Defined Auditing Process in the Enforcement of Privacy and Data Security,* Proceedings of the 1989 IEEE Symposium on Security and Privacy, pp. 206-214 (1989).

[5] J. Picciotto, *The design of an Effective Auditing Subsystem,* Proceedigns of 1987 IEEE Symposium on Security and Privacy, pp. 2-10 (1982.

[6] R. L. Rivest, *The MD5 Message Digest Algorithm*, MIT Laboratory for Computer Science and RSA Data Security, Inc., Request for Comments nr. 1321, April 1992.

[7] Axent Technologies, *Guide to Intrusion Detection*, 1995.

[8] COAST, Computer Operations, Audit and Security Technology, *www.cs.purdue.edu/ar97/research/coast.html*.

[9] Todd Atkins, *Simple Watcher*, Todd.Atkins@CAST.Stanford.edu, Stanford University.

Lockmeter: Highly-Informative Instrumentation for Spin Locks in the Linux® Kernel

Ray Bryant
raybry@us.ibm.com
IBM® Linux Technology Center
Austin, Texas

John Hawkes
hawkes@engr.sgi.com
SGI™
Mountain View, California

Abstract

Lockmeter is a tool for instrumenting the spin locks in a multiprocessor Linux kernel. Lockmeter was designed to make minimal cache disruptions, taking care to both minimize cache misses and also to minimize interprocessor cache coherency operations. Lockmeter is "highly informative" in the sense that not only does it record overall statistics for each spin lock, it also reports (where possible) these statistics on a per caller basis. This allows one to readily identify which portions of the kernel code are responsible for causing lock contention. In this paper, we describe the capabilities and implementation of Lockmeter version 1.3.

Introduction

As Linux becomes more popular for use in Web server, E-commerce and other compute intensive application environments, performance requirements on the Linux kernel will be much more stringent than for the "traditional" Linux desktop environment. This is particularly the case when Linux is used as the operating system for a symmetric multiprocessor "server" machines.

Symmetric multiprocessor (SMP) system performance is typically determined by two factors: instruction path length and lock contention. Instruction path length (that is, the time required to execute a particular function in the kernel) can be measured and optimized on a uniprocessor system using profiling tools such as SGI kernprof [SGI kernprof] or the profiling facilities of the gcc compiler. Such tools have relatively low overhead and can be used to measure realistic workloads with relatively little perturbation to the system under test.

The second factor that can limit the performance of a multiprocessor kernel is lock contention. As a general rule, correct execution of a shared-memory multiprocessor kernel requires that a lock be acquired before accessing a shared resource (e. g. a shared data structure) and released after the access. Contention arises when more than one process in the system tries to acquire a lock at the same time. Correct execution requires that only one of the processes can succeed; the other processes must be delayed until the lock is released. A delay can be implemented either by "spinning" (i. e., executing a tight instruction loop that constantly tries to acquire the lock) or by suspending the task that is attempting to access the shared resource and dispatching that processor to run some other task in the system. Locks can thus be classified as either "spin" locks or "suspend" locks depending on how a conflicting-lock access is delayed.

Each class of lock has its advantages and disadvantages:

- Acquiring or releasing a spin lock can be very inexpensive but waiting for a lock in a spin loop wastes time that could be devoted to useful work.
- A task that is suspended while waiting for a lock does not consume processor time, but the cost of acquiring or releasing a suspend lock is much higher than it is for the spin lock case.

For these reasons, both spin locks and suspend locks are typically present in a multiprocessor operating system kernel. Spin locks, however are more primitive and are normally used to implement suspend locks. In either case, excessive contention for a lock can lead to poor system performance, either because too many tasks are suspended, or because too much time is wasted spinning waiting for a lock to become available.

While instruction path length is typically straightforward to instrument without significantly perturbing the workload being measured, it is difficult to instrument lock usage in the Linux kernel without significantly impacting system performance. In Linux the code to acquire and release a spin lock is highly optimized and coded as an in-line assembler sequence. The result is that the Linux kernel can employ as few as two instructions to acquire a spin lock and one instruction to release it. Because spin lock operations are so inexpensive, they are used extensively throughout the kernel. Thus, increasing the number of instructions per lock, or, more importantly, causing an additional cache miss every time a lock is acquired or released, can significantly change system performance.

In this paper, we discuss *Lockmeter,* a tool for instrumenting the usage of spin locks in the Linux kernel. Lockmeter was developed and released as a kernel patch to the open source community by John Hawkes at SGI (Silicon Graphics) (based on a previous design by Jack Steiner at SGI). Ray Bryant of IBM subsequently enhanced this tool, and the enhanced version is now available at the SGI open-source website [SGILockmeter]. Lockmeter is supported for Intel® i386 (also called IA32 in this paper) and Alpha architectures, and will soon be supported for the Intel IA64 and MIPS architectures, although the only full implementation of Lockmeter at the moment is for IA32.

Lockmeter allows the following statistics (among others) to be measured for each spin lock:

- The fraction of the time that the lock is busy.
- The fraction of accesses that resulted in a conflict.
- The average and maximum amount of time that the lock is held.
- The average and maximum amount of time spent spinning for the lock.

This information can be used to determine which locks are causing the most contention and where in the kernel these locks are being held for excessive periods of time. Examining these statistics can help identify places in the kernel where the code may need to be restructured in order to achieve improved efficiency and performance of the Linux kernel on SMP systems.

Lockmeter is designed to minimize the number of additional cache misses and cache coherency traffic introduced by the instrumentation itself. Furthermore, Lockmeter is a raw data-gathering engine inside the kernel, leaving more computationally expensive data reduction operations to a usermode application called *Lockstat*. Lockstat retrieves the raw data from the kernel, merges it into a unified system view, and displays the results in a human-readable form.

In addition to the per lock statistics, Lockmeter also provides statistics on a per function basis. One can readily determine from the Lockmeter output not only which locks have higher than acceptable contention levels, but also determine the functions from which the highly-contended calls originated. For this reason we describe Lockmeter as being "highly informative".

In the next sections of this paper, we first describe the Lockmeter implementation. Next, we present some sample Lockmeter output. We discuss the measurements contained in that output, as well as propose an interpretation of this data. We conclude with a discussion of areas of current investigation for improving Lockmeter.

Lockmeter Implementation

Multiprocessor Linux kernels of version 2.2.x and above use two types of spin locks to serialize access to shared data. (When a Linux kernel is compiled for a uniprocessor, conditional compilation flags are set so that neither the locks nor the locking instructions themselves are present in the kernel.) The two types of locks are a mutual exclusion lock, which inside the kernel is declared by the *spinlock_t* data type, and a multiple reader, single writer "read/write lock" which is declared using the *rwlock_t* data type. The latter locks do not support promotion from reader to writer; the lock must be released in order to make this transition.

In spite of the naming convention, both of these lock types are spin locks in the classical sense – that is, if the lock cannot be immediately acquired, then the requester enters a tight spin loop checking for the lock to be released by the current owner, at which point the requester again attempts to reacquire it. If the lock is never freed, then the requester will spin forever. No attempt is made to suspend or reschedule the requester. Semaphores and wait queues are used in those cases where the hold time of the lock is sufficiently long enough to make it worthwhile to suspend the requester and schedule another process. Spinlocks and read/write locks are reserved for those cases where the lock holding time is known to be short, or as primitives to build more robust locking primitives such as semaphores.

Implementation for *spinlock_t* Locks

The kernel mutual exclusion spin locks are declared as the data type *spinlock_t* and (for IA32) are implemented as a structure consisting of a single 32-

bit word. To acquire the lock one calls *spin_lock();* to release the lock one calls *spin_unlock().* There are variations of these calls that save and restore the interrupt state of the machine, but for the purposes of this paper we will ignore these variations (and will ignore them for the *rwlock_t* case below) since they consist of a wrapper that surrounds the *spin_lock()* or *spin_unlock()* call itself. Since the Lockmeter implementation modifies these basic macros, it modifies the variations as well.

In a non-lockmetered kernel, *spin_lock(), spin_unlock(),* and *spin_trylock()* are C language macros that resolve to gcc *inline* functions that contain assembler instructions that implement the lock operations. The Lockmeter patch renames these inline lock macros with a *nonmetered_* prefix (e.g., *nonmetered_spin_lock()*), then declares new *spin_lock()* (*et al*) macros that call external lockmetering routines *_spin_lock_()* and *_spin_unlock_().* Obviously, procedure calls are more expensive than a few inline assembler instructions, but the instrumented lock code is too large to be inserted inline.

The *nonmetered_*()* primitives are employed by *_spin_lock_()* and *_spin_unlock_()* to implement the instrumented locks. Thus the Lockmeter code is largely machine (and lock implementation) independent, since it uses the existing underlying lock primitives.

As an example of how this is done, consider the following pseudo-code implementation of *_spin_lock_():*

```
void __spin_lock_(spinlock_t *lock) {
    if (nonmetered_spin_trylock(lock)) {
            /* we acquired the lock without waiting */
            update statistics for this case
    } else {
            /* we must to spin to wait for the lock */
            record time we started spinning
            nonmetered_spin_lock(lock);
            record time we stopped spinning
            update statistics for this case
}
```

The key data structures used by the Lockmeter spin lock routines are:

- The *directory*, which is organized as a hash table keyed by the address that invokes the instrumented lock routine, and further indexed by lock type.
- The *count* array, which is a two-dimensional array indexed by hash-table index and by logical CPU number.

The hash table entries in the *directory* contain no lock statistics data; all that is present is the calling address for the instrumented lock routine, the address of the lock itself, and a lock type field. Separate lock types exist for *spinlock_t* locks, *rwlock_t*'s acquired for read mode, and *rwlock_t*'s acquired for write mode. The lock type information is used as part of the hash lookup and by the data reduction program, *Lockstat.*

When a hash lookup is performed, the *directory* entry information is used to resolve hash table synonyms. The result of the hash table search is an index that is used in combination with the current logical CPU number to select an entry in the *count* array where the actual statistics are stored. This makes the *directory* a performance-efficient read-mostly structure for all processors. The *directory* is only updated when each lock request is first encountered, making them relatively rare events once the system is running and the *directory* has been populated with the frequently used spin locks. A single nonmetered spin lock protects *directory* updates.

A *directory* entry is either a "single-address" entry or a "multi-address" entry. A single-address entry contains the address of the *spin_lock()* caller and the address of the *spinlock_t* structure being locked. Such an entry results when a given caller always specifies the same lock address. Lock requests for statically allocated locks are typically of this type. A multi-address entry results when a particular lock request specifies different lock addresses on different invocations. Typically this happens when a lock routine is acquiring a *spinlock_t* that is part of a larger dynamically allocated structure.

Each new *directory* entry is initially allocated as a single-lock entry. It converts to a multi-lock entry if a search of the hash table finds a match on the caller's address, but the current lock address does not match the value previously associated with that caller. At that point, the entry converts to a multi-lock entry by setting the lock address in the entry to zero. The data reduction program, Lockstat, recognizes this as a special case and reports only a coalesced summary of all locks set by this caller's address. For the more common single-lock entries, Lockstat reports per caller statistics for each unique spin lock address.

Each occupied *directory* entry contains a unique index into the *count* array. Each *count* entry represents a spin lock address, and the entry contains the count of lock requests for that specific spin lock, the cumulative sum of "hold" times and "wait" times for that lock, and the maximum observed "hold" and "wait" times. Thus, each time a spin lock is acquired and each time it is released, the lockmetering routines update a *count* entry.

An important implementation efficiency is to give each CPU its own private *count* array. One advantage this provides is that we do not need to protect concurrent *count* entry updates with a (nonmetered!) spin lock. Another advantage is that processor-private updates only perturb a single processor's cache, and thus colliding updates do not cause expensive inter-processor cache references.

Each entry of the *count* array maintains lock statistics for one *spin_lock()* call on a particular processor. (We refer to these as "per caller" statistics.) Lockstat calculates global statistics for a lock by aggregating all of the single address statistics entries that correspond to the same lock. Statistics for the multi-address entries are reported on a per-caller basis only.[1] Experience has shown that eliminating the distinction between single and multi-address entries (by hashing on the caller address and the lock address, for example) clutters the Lockmeter output with many temporary, single-use locks.

In the *_spin_lock_()* routine, the *count* array is updated as follows:

1. Look up the calling address in the *directory*.
2. Using the directory *index,* find the *count* entry and update acquire-time statistics such as number of times the lock has been requested and the spin time.
3. Store the acquisition timestamp in the *count* entry as the first step in calculating the lock "hold" time.

At unlock time we need to locate this same *count* entry in order to finish the "hold" time calculation, which means we need to determine the *directory* index of that entry. To compute the *directory* hash index, we would need the address of the previous *_spin_lock_()* call, but that is presumably unknown to us at *_spin_unlock_()* time. We could remember these *directory* indices in yet another per-processor data structure, but this would introduce more code to manage the data structure and potentially more costly cache misses.

We solve this problem by storing the hash index in the lock itself. In every implementation we have seen, a *spinlock_t* is allocated as a 32-bit location, but few of the 32 bits are actually used. We therefore can use some of the remaining bits to save the hash index that was computed in *_spin_lock_()* for later use at *_spin_unlock_()* time. Since the lock location is already in the cache of the local processor due to acquiring the lock, storing the hash index in the lock results in negligible additional overhead and no additional cache misses.

Implementation for Read Locks

As with *spin_lock()* and *spin_unlock(),* the non-instrumented multiprocessor kernel implements *read_lock()* and *read_unlock()* as inline functions that generate only a few assembly language instructions each. Lockmeter replaces these inline functions with calls to *_read_lock_()* and *_read_unlock_(),* respectively. The existing lock primitives are renamed to nonmetered versions, just as for the *spinlock_t* case, and the nonmetered versions are utilized by the instrumented lock routines to perform the actual lock operations.

In *_read_lock_(),* acquire time statistics such as the count of requests, the time spent spinning waiting for the lock, and the count of times that the lock was obtained without spinning are maintained in the *count* array at the appropriate hash index. The *directory* entry is set to lock type "*rwlock_t* acquired in read mode." As before, separate statistics entries are maintained for each processor. Thus we record per caller statistics for read locks for statistics that are completely known at read-lock acquire time.

Hold-time statistics for read locks are problematic, however, since more than one processor can concurrently hold the same read lock. Therefore, *_read_unlock_()* cannot easily determine which *_read_lock_()* it is releasing, and this makes direct calculation of the read lock hold time impossible. (Since a *spinlock_t* is a mutual exclusion lock, each *_spin_unlock_()* ends the hold time that began with the most recent *_spin_lock_()* for this lock, so determining the hold time is straightforward.) One could keep track of the correspondence between each *_read_lock_()* and its matching *_read_unlock_()* using a list associated with each processor, but maintaining this list would be expensive and of questionable analytic value.

[1] It is possible for a lock to be accessed both by static and dynamic lock requests; in this case the lock statistics will be split between global and per-caller statistics. We are not aware of any lock usage in the kernel that follows such a pattern.

Lockmeter solves these problems as follows. We first require that the *rwlock_t* be declared as a pair of 32 bit words. (The Linux 2.3.99-pre6 kernel uses the 2nd word as a debug flag; Lockmeter merely requires that this debug word be present. We cannot store any data in the first word of the lock structure since in the current implementation of read/write spin locks there are no unused bits in that word.)

The first time a read or write lock is acquired on a variable of type *rwlock_t,* a read lock index is allocated for the lock. This is done by incrementing a global variable and assigning the next available index to this lock variable. The read lock index is stored in part of the second word of the lock for use on subsequent lock operations. The read lock index specifies the location in the *read_lock_count* array, where hold time statistics for this lock are stored.

Like the *count* array for *spinlock_t* metering, the *read_lock_count* array is a two-dimensional array; the first index being the read lock index, the second index being the current logical CPU number. The *read_lock_count* array contains running sum and count information necessary to calculate average read lock hold times as well as the overall read lock utilization time. Once again, statistics for each lock are maintained in storage private to each processor, and the data reduction program, Lockstat, is responsible for merging these statistics across processors.

However, for read locks, the data is not kept on a per caller basis, since the read lock index is the same for all users of the lock. Per caller statistics for read lock hold times would require that we match read lock and unlock operations using a per processor lock list, and we regard that approach as too expensive to employ.

The running sum for lock hold times in the *read_lock_count* structure are updated as follows based on an idea from [IBM1, IBM1A]:

At lock acquire: *running_sum -= get_cycles64();*
At lock release: *running_sum +=get_cycles64();*

(Here *get_cycles64()* returns the current 64 bit Time Stamp Counter (TSC register) value.) This approach keeps us from having to maintain the lock acquire time for each processor for each read lock (and it keeps us from having to deal with recursion problems related to a read lock that is set more than once by a particular processor).

The above works because the above is equivalent to the more straightforward algorithm, where at acquire time we record a timestamp:

lock_acquire_time = get_cycles64();

And at release time we decrement the current timestamp from the saved acquisition timestamp:

running_sum += get_cycles64()
– lock_acquire_time;

(provided that *lock_acquire_time* is kept on a per processor basis.)

This calculation can alternatively be done as:

running_sum += get_cycles64(); /* (1) */
running_sum -= lock_acquire_time; /* (2) */

And since addition is commutative, we will get the same result if we do *(2)* before *(1)*. If we do (2) first, we might as well do it at lock request time and avoid the temporary variable:

running_sum -= get_cycles64(); /* (2) */

then at lock release time, we do (1):

running_sum += get_cycles64(); /* (1) */

which is how we described the calculation originally.

However, we do have the problem that the *running_sum* is only correct when there are no read lock holders. Otherwise, the *running_sum* is a negative number (provided we assume that the maximum read lock hold time is very small compared to the current *get_cycles64()* value). Complete details of how this is done can be found in the source code [SGILockmeter] and [IBM2]. For this paper it will suffice to say that read lock hold time statistics are enabled and disabled on a per lock basis, and a transition from enabled to disabled state is only allowed when there are no read lock holders of the lock.

The last part of the read lock statistics to be discussed here is read lock utilization; that is, the fraction of time that a particular *rwlock_t* is owned in read mode by one or more readers. This is done by maintaining in the *read_lock_count* array a running sum of the busy period lengths that ended on the current processor for this lock. (A busy period is defined as the time starting when the number of read lock holders for a lock transitions from zero to one, until the next time that the number of readers transitions from one to zero.) It is easy enough for the instrumented read lock routines to recognize the start and end of a busy period; the hard problem is how to update the running sum of busy period lengths without using a global variable. Using a global variable would cause too much interprocessor cache coherency traffic as well as requiring a locked update of some kind to deal with concurrent accesses.

The solution here [IBM3] is to maintain in the *read_lock_count* entry for this processor (and this lock) the last time (measured using *get_cycles64()*) that a busy period started due to a *_read_lock_()* call executed on this processor for this lock. Also, when a busy period starts, the current logical CPU number is stored in the *rwlock_t* structure. (Since this structure is pulled into the local cache when the lock is acquired, this operation causes minimal additional overhead.)

When a processor detects the end of a busy period in *_read_unlock_()*, that processor can determine the busy period's start time by looking in the appropriate *read_lock_count* array entry for the processor that started the busy period; the logical CPU number of that processor is obtained from the *rwlock_t* structure. From this and the current time, we know the busy period length. The busy period length is then added to a running sum of busy period lengths, and the number of busy periods is incremented. (Both of these variables are stored in per processor storage in the *read_lock_counts* array.) This approach results in at most one remote cache reference at the end of each busy period.

Implementation for Write Locks

Since write locks are mutual-exclusion locks, the implementation of metering write mode locks on a *rwlock_t* is basically the same as it is for *spin_lock()*. The only difference is that the hash table index for the lock statistics entry is saved in the *read_lock_counts* array instead of in the lock itself. This allows us to keep per-caller statistics for write mode locks on a *rwlock_t*.

When the *directory* entry is being searched, the hash function is based upon the return address of the *_write_lock_()*. The *directory* entry is set to lock type of "*rwlock_t* acquired in write mode".

Each *rwlock_t* in the kernel thus can have three statistics structures associated with it:

1. The global read lock counts structure in the *read_lock_count* array.
2. A statistics entry containing read lock spin wait times on a per caller basis in the *count* array. This entry is updated only at *read_lock()* time with statistics that are known at the time that the read lock is acquired.
3. A statistics entry containing write lock spin and hold times on a per caller basis in the *count* array. This entry is updated at *write_lock()* and *write_unlock()* times.

The Lockstat Program

Lockstat is the user interface to the Lockmeter facility. It implements the following categories of operations:

- Instrumentation control: Lockmeter statistics can be enabled, disabled, queried, reset, or released under control of the Lockstat program. ("release" frees the kernel storage occupied by the *count*, *directory*, and *read_lock_count* data structures.)
- Data reduction: Lockmeter statistics can be read from the kernel and either saved for later data reduction or immediately reduced and printed. Either function can be performed on-demand or automatically using a periodic timer. During data reduction, Lockstat uses the kernel's *System.map* file to translate lock names and function addresses into symbolic names.

A typical measurement scenario would look something like the following:

```
# begin measurement
lockstat on
# run experiment
...
# end measurement
lockstat off
# reduce data
lockstat –m System.map print > lock.report
# clear statistics to prepare for next measurement
lockstat reset
```

Alternatively, one could use the "lockstat get" command to fetch the raw lockmeter statistics for printing at a later time.

Lockstat supports the notion of multiple measurement "intervals". Each execution of "*lockstat on*" begins a new measurement interval; executing "*lockstat off*" ends the interval. Statistics from each interval are merged together during data reduction. The "reset" command is used to clear data from previous measurement intervals and begin a new set of measurements. This facility allows one to run several repetitions of an experiment and have Lockstat merge the statistics from the repetitions into a single Lockstat report.

Additional documentation on Lockstat is provided by the "*--help*" option to Lockstat; of course the source code itself is available at [SGILockmeter].

Lockstat Output

The Lockstat report consists of three major sections: "SPINLOCKS", "RWLOCK READERS" and "RWLOCK WRITERS." Each of these sections is subdivided into per-lock statistics entries and per-caller statistics entries. A condensed version of a Lockstat report is provided in the Appendix and is discussed in the following. Due to space constraints, this condensed report provides statistics for only a small fraction of the locks present in a real Lockstat report. Also, in order to make the report fit onto a printed page, we have removed some of the columns of Lockstat output.

The top of the report shows information about the system being measured. (For this report the system was an IBM Netfinity® 7000 M10 Intel® Pentium® II Xeon™ 4-way SMP system.) This particular Lockstat report was generated for all 4 processors. However, since Lockmeter statistics are kept on a per processor basis, one could also generate a report based on a single processor's lock usage.

The data presented here summarizes Lockmeter statistics recorded during an experiment of approximately 140 seconds in length. The workload being run for this experiment is Volanomark™ [VMark], a benchmark written in the Java™ language. For these measurements, Volanomark was run using the IBM® Developer Kit for Linux®, Java™ Technology Edition, Version 1.1.8. For further details about this benchmark environment, see [JTThreads, SMPPerf]. For discussion purposes here, it is sufficient to know that the benchmark is CPU bound and causes a large number of Linux processes (threads) to be created and scheduled.

"SPINLOCKS" Section of the Report
The SPINLOCKS section of the Lockstat report shows statistics information for the *runqueue_lock* and the *timerlist_lock* (see Note 1 and Note 2 in the report). These locks protect the scheduler queue and the timer list data structures, respectively.

The first line of the report for each lock shows overall statistics, while subsequent lines provide per-caller statistics for that lock. At the end of the SPINLOCKS section (see Note 2) are some multilock statistics entries. (These entries can be recognized because there is no lock name associated with this part of the report.) These entries report on *spin_lock()* calls that request more than one lock address during the measurement run; typically this means that the locks being set are dynamically allocated. Rather than report on each individual lock, Lockstat aggregates all such requests and reports them only by lock caller. As previously discussed, this avoids cluttering the Lockstat output (and the kernel Lockmeter statistics) with information about many temporary, single-use locks.

The first column of the SPINLOCKS report (labeled UTIL) is the lock utilization.[2] This is defined as the fraction of time that the lock was held during the report interval. The second column (CON) is the fraction of lock requests that found the lock was busy when it was requested. The third and fourth columns (HOLD MEAN and MAX) show the mean (average) and maximum hold times for the lock. The next two columns (WAIT MEAN and MAX) give the mean and maximum times that a lock requester had to wait to obtain a lock.[3] The next column of the report (TOTAL) gives the total number of requests that occurred for the lock during the measurement interval. In a full Lockstat report, subsequent columns display the number of requests that had to spin for the lock; this and similar columns have been removed from the report here in order to conserve space. The last column gives either the lock name (*runqueue_lock* for the case at Note 1) or the calling location, as appropriate.

"RWLOCK READERS" Section of the Report
The RWLOCK READERS section of the Lockstat report provides statistics about *read_lock()* requests for *rwlock_t* locks. Like the SPINLOCKS section, this section is divided into a lock statistics section and a multilock section; to conserve space the latter has been omitted from this condensed report. We show here entries for the *tasklist_lock* and the *xtime_lock*. The *task_list* lock is held in read mode to scan over all tasks in the system, while the *xtime_lock* is held in read mode to read system time information.

Just as the previous report, the UTIL column gives the lock utilization. For read-mode locks, this is defined as the fraction of time that there is at least one reader for the lock. Total utilization for this lock is the sum of the utilizations from the "RWLOCK READERS" and "RWLOCK WRITERS" sections of the Lockstat report. The "HOLD MEAN" column gives the average time that the lock was held in read

[2] The UTIL field can optionally be replaced by a rate request field via a Lockstat command-line option.
[3] The mean wait time is defined as the average over all requesters that had to wait, rather than over all requesters.

mode, averaged over all readers of the lock. Given the current implementation Lockmeter, this statistic is only available on a global and not a per-caller basis.

The "MAX READERS" column gives the maximum number of readers that simultaneously held the lock at any one time. The fact that this number is 5 on a 4-way SMP system indicates that some processor holds this lock more than once. One possible explanation for this is that the timer-interrupt code obtains a read lock on *task_list* lock; if all four processors in the system already hold the *task_list* lock and then a timer-interrupt occurs, this will cause one of the processors to re-lock the *tasklist_lock.*

The "RDR BUSY PERIOD" columns provide mean and maximum times of busy periods for the lock. A busy period for a read lock is defined as the period of time starting when the number of read-lock holders goes from zero to one and ending when there are no read-lock holders of the lock. Busy period information tells us how long a write-requester might have to wait in order to obtain the lock. The sum of the lengths of busy periods is used to calculate the read-lock utilization.

"RWLOCK WRITERS" Section of the Report
The "RWLOCK WRITERS" section of the Lockstat report provides statistics about *write_lock()* requests for *rwlock_t* locks. The same format and locks discussed in the "RWLOCK READERS" section above are used here.

The utilization column (UTIL) in this section of the report gives the fraction of time that the lock was held in write mode. The wait-time statistics for a write-mode lock are given as the mean and maximum wait time over all requests as well as the mean and maximum wait times for write-lock requesters who had to wait due to other write-lock holders of the lock (as opposed to write-lock requesters who had to wait due to other read-lock holders of the lock). These statistics are given in the "WAIT (ALL)" and "WAIT (WW)" columns of the report, respectively. Similarly, the "SPIN ALL" and "SPIN (WW)" columns of the report give the count of requesters who had to spin for the lock and the count of requesters who had to spin for the lock due to another write-lock holder.

Analysis of the Lockstat Data

As an example of the kind of analysis one might do with the Lockstat data, we now discuss the results reported by Lockstat for this benchmark.

Inside the Linux kernel, the *runqueue_lock* protects access to the scheduler queue. Since the Volanomark benchmark is a scheduler intensive benchmark [JTThreads, SMPPerf] we expect to see contention for the *runqueue_lock* in this report. At *Note 1* we see that utilization of this lock is nearly 22% and that 25% of the requests for this lock had to spin-wait for the lock. One can see that most of the time spent holding this lock was due to requests that occurred at *schedule+0xd0*. From the numbers reported here, one can see that this caller was responsible for 40% of the requests to this lock and 67% of the utilization. This caller also held the lock for longest time (average and maximum). Examination of the kernel/sched.c source code shows that this lock is held from entry of *schedule()* until after the goodness calculation has completed. Previous work [JTThreads, SMPPerf] has shown that for this benchmark, the goodness calculation can consume a significant amount of CPU time. These lock statistics for the *runqueue_lock* validate that observation.

Note also how the hold times due to *schedule+0xd0* impact the other lock requests. See, for example, the request at *schedule+0x444*. While the utilization of the lock due to this request is only 1.7%, 56% of all such lock requests had to spin for the lock. This indicates that although the lock holding time for this request is low, usage of the lock elsewhere causes contention and results in an average 14 microsecond delay and a maximum delay of 285 microseconds. Examination of the source code shows that this lock request is from the inlined function *__schedule_tail()* just before the scheduler returns, running a new process. Hence the 14 microsecond mean wait time here directly affects the latency of the system in starting to run a new process.

Similarly, if we examine the *tasklist_lock* (see *Note 5* in the Lockstat report) we see that *do_fork()* (the worker routine that implements the fork system call) was delayed an average of 7.3 microseconds and a maximum of over 1.5 milliseconds waiting for the *tasklist_lock.* Note that it could have been worse, since the maximum read-lock busy period for this lock was over 8 milliseconds (*Note 4*).

Of course, these observations are correct only for this particular benchmark and are not necessarily indicative of actual bottlenecks in the Linux kernel.

Rather they are provided as examples of where potential bottlenecks might occur and how the Lockstat report can be used to find such problems.

Concluding Remarks

As the Linux kernel continues to be deployed on large SMP servers, additional performance optimization of the kernel will be required in order to achieve performance comparable to the more mature operating systems that have traditionally been used on such hardware. As systems become larger and more complex, we will need increasingly powerful tools to analyze and diagnose system performance problems. Lockmeter endeavors to be one such measurement tool in a Linux performance toolbox that can be used to diagnose and optimize Linux system performance.

We are continuing to improve Lockmeter and Lockstat. Alternative implementations of the read/write lock statistics recording mechanism are currently under investigation, as are benchmark studies to estimate the Lockmeter measurement overhead. Such overhead measurements should be available at the time of the 2000 Atlanta Linux Showcase and Conference. An updated version of this paper should also be available at that time on the SGI lockmeter website [SGILockmeter] or the IBM Linux Technology Center website [IBMLTC].

References

[SGI Kernprof]: "Kernel Profiling," http://oss.sgi.com/projects/kernprof

[SGI Lockmeter]: "Kernel Spinlock Metering for Linux." http://oss.sgi.com/projects/lockmeter

[IBM1] United States Patent 5,872, 913, "System and Method for Low Overhead, High Precision Measurements using State Transitions," Robert F. Berry et al.

[IBM1A] United States Patent 5,920,689, "System and Method for Low Overhead, High Precision Measurements using State Transitions," Robert F. Berry et al.

[IBM2] "Efficient Update of Shared Resource Usage Statistics," IBM Technical Disclosure Bulletin, *to appear.*

[IBM3] "Careful Update and Control of Hold Time Statistics for Shared Multiuser Resources," IBM Technical Disclosure Bulletin, *to appear.*

[Volano]: "Volano Java Chat Room," Volano LLC. http: //www.volano.com.

[JTThreads]: "Java technology, threads, and scheduling in Linux--Patching the kernel scheduler for better Java performance," Ray Bryant, Bill Hartner, IBM. http://www4.ibm.com/ software/developer/library/java2 /index.html.

[SMPPerf]: "SMP Scalability Comparisons of Linux® Kernels 2.2.14 and 2.3.99," Ray Bryant, Bill Hartner, Qi He, and Ganesh Venkitachalam, Proceedings of the 2000 Atlanta Linux Showcase and Conference, Atlanta, Ga., October, 2000.

[IBMLTC] Linux Technology Center, http: //oss.software.ibm.com/developerworks/ opensource/linux/.

Trademark and Copyright Information

Appendix: (Condensed) Lockstat Output

```
System: Linux testlinux.austin.ibm.com 2.3.99-pre6 #147 SMP Tue Aug 22 15:18:05 CDT 2000 i686

All (4) CPUs

Start time: Fri Aug 25 16:09:05 2000
End   time: Fri Aug 25 16:11:26 2000
Delta Time: 141.33 sec.
Hash table slots in use:      356.

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
SPINLOCKS         HOLD            WAIT
   UTIL    CON    MEAN( MAX )  MEAN ( MAX  ) TOTAL  NAME
. . .
  21.86% 25.12% 3.3us( 87us) 4.4us(311us) 9480279 runqueue_lock                        ←Note 1
   1.62% 25.14% 3.3us( 13us) 1.1us(136us)  696844  __wake_up+0x110
   0.00%  0.00% 0.2us(0.2us)   0us              1  __wake_up_sync+0xfc
   0.00% 23.60% 5.1us( 19us) 4.3us(112us)     322  process_timeout+0x1c
   0.00%  8.71% 0.5us(1.3us) 0.2us(7.1us)     551  release+0x28
   0.00% 61.11% 2.0us(5.1us) 1.2us(5.9us)      36  schedule_tail+0x48
  14.69% 19.63% 5.5us( 87us) 1.7us(311us) 3806754  schedule+0xd0
   1.71% 56.33% 2.1us( 12us)  14us(295us) 1150208  schedule+0x444
   0.51% 14.35% 4.3us( 28us) 0.5us( 51us)  165306  schedule+0x710
   0.68% 12.89% 0.8us(4.5us) 0.7us(178us) 1224206  send_sig_info+0x2a0
   0.00% 40.57% 4.8us( 10us) 0.9us( 11us)     801  setscheduler+0x68
   0.78% 46.40% 0.9us(6.1us)  15us(265us) 1210679  sys_sched_yield+0xc
   1.88%  5.56% 2.2us( 14us) 0.4us(164us) 1224571  wake_up_process+0x18
. . .
   0.95%  3.87% 0.6us( 17us) 0.1us(9.4us) 2380970 timerlist_lock                       ←Note 2
   0.13%  4.66% 0.4us(4.1us) 0.1us(9.4us)  405952   add_timer+0x14
   0.33%  4.50% 0.5us(4.3us) 0.1us(9.2us) 1004500   del_timer+0x14
   0.00%  0.18% 0.5us(1.5us) 0.0us(2.1us)     565   del_timer_sync+0x20
   0.31%  3.35% 0.6us(4.2us) 0.0us(8.2us)  777490   mod_timer+0x18
   0.03%  1.86% 3.0us( 17us) 0.0us(4.8us)   14134   timer_bh+0x12c
   0.16%  0.99% 1.2us(5.1us) 0.0us(6.3us)  178329   timer_bh+0x2b4
. . .
   3.47%  0.00% 7.0us(142us) 0.0us(7.1us) 702015 __wake_up+0x24                        ←Note 3
   0.22%  0.56% 0.4us( 36us) 0.0us(5.9us) 811287 dev_queue_xmit+0x30
   0.01%  0.00% 1.1us(5.9us)   0us         14558 do_IRQ+0x40
   0.01%  0.00% 4.7us( 31us)   0us          1521 do_brk+0x108
   0.00%  0.00% 0.2us(0.9us)   0us           429 do_exit+0x240
   0.00%  0.00% 0.2us(0.6us)   0us           338 do_fork+0x6fc
. . .
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
RWLOCK READERS HOLD    MAX   RDR BUSY PERIOD    WAIT
  UTIL  CON    MEAN  READERS MEAN    (   MAX  ) MEAN ( MAX )   TOTAL NAME
. . .
  52.91% 0.00% 105.8us   5   114.8us (8274.8us) 0.0us(3.3us) 1402747 tasklist_lock    ←Note 4
         0.00%                                    0us             28  count_active_tasks+0x10
         0.23%                                  0.0us(2.3us)     429  exit_notify+0x1c
         0.00%                                    0us              5  exit_notify+0xb8
         0.00%                                    0us            576  get_pid_list+0x18
         0.00%                                  0.0us(3.0us) 1224079  kill_something_info+0xb8
         0.00%                                    0us          11002  proc_pid_lookup+0x4c
         0.00%                                    0us              7  proc_root_lookup+0x30
         0.00%                                    0us         165306  schedule+0x6d0
         0.00%                                    0us             25  session_of_pgrp+0x14
         1.12%                                  0.0us(3.3us)     801  setscheduler+0x78
         0.00%                                    0us             18  sys_setpgid+0x38
         0.00%                                    0us              1  sys_setsid+0x10
         0.00%                                    0us            461  sys_wait4+0x158
         0.00%                                    0us              9  will_become_orphaned_
                                                                        pgrp+0x14
. . .
 0.33%   0.06%    0.5us   2     0.5us   (     6.5us) 0.0us(528us)   856780 xtime_lock
         0.06%                                  0.0us(528us)   856780  do_gettimeofday+0x10
. . .
```

Appendix: (Condensed) Lockstat Output (continued)

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
RWLOCK WRITERS   HOLD          WAIT (ALL)     WAIT (WW)        SPIN SPIN
UTIL     CON  MEAN ( MAX ) MEAN ( MAX  ) MEAN (  MAX ) TOTAL ALL  WW  NAME

0.00% 10.53% 0.8us(2.7us) 9.8us(1515us) 0.8us( 1.7us)  1691 173   5 tasklist_lock
0.00% 10.68% 1.2us(2.6us) 7.3us(1515us) 0.8us( 1.7us)   833  84   5  do_fork+0x8a4       <-Note 5
0.00%  2.80% 0.2us(0.8us) 0.2us(  13us)   0us           429  12   0  exit_notify+0x284
0.00% 17.95% 0.7us(2.7us)  25us( 486us)   0us           429  77   0  release+0x78
. . .
0.12%  1.29% 6.2us(802us) 0.0us(7.7us)  0.9us( 4.3us) 28352 281  84 xtime_lock
0.03%  1.68% 2.8us(802us) 0.0us(7.7us)  1.0us( 3.1us) 14193 180  59  timer_bh+0x14
0.10%  0.89% 9.6us( 24us) 0.0us(6.4us)  0.8us( 4.3us) 14159 101  25  timer_interrupt+0x14
```

The Elements of Cache Programming Style

Chris B. Sears

Google Inc.

Those who cannot remember the past are condemned to repeat it - George Santayana

1. Introduction

Cache memories work on the carrot and stick principle. The carrot is the Principle of Locality and the stick is Amdahl's Law. The Principle of Locality says that programs tend to cluster their memory references. A memory location referenced once is likely to be referenced again: temporal locality. A memory location nearby a referenced location is likely to be referenced soon: spatial locality. And Amdahl's Law says that the performance improvement to be gained from using some faster component is limited by the fraction of time the faster component is used. In this case, CPU and cache are fast components and memory is slow.

If your program adheres to the Principle of Locality, it benefits from fast cache memory and runs at processor speed. If it doesn't, it is held accountable to Amdahl's Law and runs at memory speed. Hit rates have to be very high, say 98%, before incremental increases in processor speed are even noticeable.

Amdahl's Law has a special circumstances penalty for multiprocessors [Schimmel94]. Thrashing on a multiprocessor can slow down all of the processors. They each wait for each other waiting for memory and the leverage a multiprocessor offers works in reverse. Adherence to the Principle of Locality for multiprocessors, but not to the point of False Sharing, isn't just a nicety, it is a necessity.

The object of cache programming style is to increase this locality. It is important to understand the structure and behavior of caches, but it is more important to know the basic properties to take advantage of and the worst cases to avoid. This article goes into details and the summary provides guidelines.

2. An Example

As a running example, I am going to look at Linux [Maxwell99] and at the scheduler in particular. The idea is to modify data structures and code just slightly, trying to use the cache more effectively. Hopefully I will achieve two goals: a practical tutorial on caches and some performance improvements for Linux.

Instead of talking about cache memory systems in general, I will mostly use my circa 1998 350 MHz Deschutes Pentium II system as a specific example. It has these characteristics:

Storage	Size	Latency	Notes
register	32 bytes	3 ns	register renaming file
L1	32 K	6 ns	on-chip, half Pentium-II clock rate
L2	256 K	57 ns	off-chip, on-package [Intel99a]
memory	64 MB	162 ns	100 MHz SDRAM, single bank
disk	10 GB	9 ms	DMA IDE
network	whatever	whenever	56K PPP

Figure 1. Storage hierarchy sizes and latencies

These numbers are subject to change. CPU performance improves at about 55%/year and memory improves at about 7%/year. Memory is big, cheap and slow while cache is small, fast and expensive. Double Data Rate SDRAM and Rambus, when available, will improve memory bandwidth but not latency. These improvements will help more predictable applications like multimedia but not less predictable programs such as Linux.

3. Cache Basics

First, a few words about caches in general. Cache memory fits into the storage hierarchy in terms of both size and speed. Cache line misses, page faults and HTTP requests are the same thing at different levels of this hierarchy. When a Squid proxy doesn't have an object in its cache, it forwards the HTTP request to the origin server. When a CPU requests an address which isn't in memory, a page fault occurs and the page is read from disk. When a CPU requests an address which isn't in cache, the containing cache line is read from memory. LRU, working set, associative, coherency, hashing, prefetching are all techniques and terminology which are used in each level of the storage hierarchy.

In each case, one smaller faster level in the hierarchy is backed by another bigger slower level. If performance is limited by excessive use of the slower level, then according to Amdahl's Law, little improvement can be made by just making the faster level faster.

With respect to cache memory [Handy98], the most important thing to understand is the cache line. Typically a cache line is 32 bytes long and it is aligned to a 32 byte offset. First a block of memory, a memory line, is loaded into a cache line. This cost is a cache miss, the latency of memory. Then, after loading, bytes within a cache line can be referenced without penalty as long as it remains in the cache. If the cache line isn't used it will be dropped eventually when another memory line needs to be loaded. If the cache line is modified, it will need to be written before it is dropped.

This is the simplest and most important view of a cache memory. Its lesson is two-fold: pack as much into a cache line as possible and use as few cache lines as possible. Future memory bandwidth increases (DDR and Rambus) will reward this practice. The more complex characteristics of cache, the structure and behavior, are important for understanding and avoiding worst case cache behavior: thrashing.

Competing for and sharing of cache lines is a good thing, up to a point, when it becomes a bad thing. Ideally a fast cache will have a high cache hit rate and the performance will not be bound to the speed of the memory. But a really bad thing, thrashing, happens when there is too much competition for too few cache lines. This happens in worst case scenarios for data structures. Unfortunately the current profiling tools look at the instructions rather than data. This means that a programmer must be aware of worst case scenarios for data structures and avoid them. A useful tool for finding a hot spot is cacheprof [Seward].

4. The Pentium II L1 and L2 Caches

The Pentium II [Shanley97] 32K L1 cache consists of 1024 32 byte cache lines partitioned into instruction and data banks of 512 lines each. It uses the color bits 5-11 to index into an array of sets of cache lines. In parallel, it compares the tag bits 12-31 (12-35 with Pentium III Physical Address Extension) for each of the cache lines in the indexed set. L1 uses a 4-way set associative mapping which divides the 512 lines into 128 sets of 4 cache lines.

Each of these sets is really a least recently used (LRU) list. If there is a match, the matching cache line is used and it is moved to the front of the list. If there isn't a match, the data is fetched from L2, the cache line at the end of the list is replaced and the new entry is put at the front of the list.

Two memory lines of the same color compete for the same set of 4 L1 cache lines. They are off the same color if their color bits (5-11) are the same. Alternatively they are of the same color if their addresses differ by a multiple of 4096: 2 ^ (7 color bits + 5 offset bits). For example, address 64 and 12352 differ by 12288 which is 3*4096. So, 64 and 12352 compete for a total of 4 L1 cache lines. But 64 and 12384 differ by 12320, not a multiple of 4096, so they don't compete for the same L1 cache lines.

Instructions are also cached. The Pentium II L1 cache is a Harvard, or split instruction/data cache. This means that instructions and data never compete for the same L1 cache lines. L2 is a unified cache. Unified means that there is a single cache bank and that instructions and data compete for cache lines.

L2 is similar to L1 except larger and much slower. The on-package 256K L2 cache on my Pentium II has 8192 cache lines. It is also 4-way set associative but is unified. There are Pentium II's with 512K of L2 which increase the set size to 8. Also, there are PIII's with up to 2 MB of L2. If there is a cache line miss for L2, the cache line is fetched from memory. Two memory lines compete for the same L2 cache lines if they differ by a multiple of 64K: 2 ^ (11 cache color bits + 5 offset bits).

The important things to remember about my Pentium II are:

- cache lines are 32 bytes in size and are aligned to 32 byte offsets
- memory locations which are offset by multiples of 4K bytes compete for 4 L1 cache lines
- memory locations which are offset by multiples of 64K bytes compete for 4 L2 cache lines.
- L1 has separate cache lines for instructions and data - Harvard
- L2 shares cache lines between instructions and data - unified

5. Variable Alignment

We will start with the simple stuff. It is better to align just about everything to a long word boundary. Linux is written in the gcc programming language and a careful study of the gcc standards document, "Using and Porting GNU CC" [Stallman00], is therefore necessary: no one embraces and extends quite like Richard Stallman. gcc is particularly helpful with structure field alignment which are intelligently and automatically aligned. ANSI C Standard allows for packing or padding according to the implementation.

```
struct dirent {
    long            d_ino;
    __kernel_off_t  d_off;
    unsigned short  d_reclen;
    char            d_name[256];
};
```

Figure 2. <linux/dirent.h>

gcc automatically aligns d_reclen to a long boundary. This works well for unsigned short, but for short on the x86 the compiler must insert sign extension instructions. If you are using a short to save space, consider using an unsigned short. For example, in <linux/mm.h> changing the field vm_avl_height into an unsigned short saves 32 bytes of instructions for a typical build. It could just as well be an int.

```
struct vm_area_struct {
    ...
    short           vm_avl_height;       // unsigned short or int
    ...
};
```

Figure 3. struct vm_area_struct

Strings should be aligned as well. For example, strncmp() can compare two long words at a time, cheap SIMD, if both source and destination are long word aligned. The x86 code generator for egcs 2.95.2 has a nice little bug that doesn't align short strings at all and aligns long strings to the cache line:

```
char* short_string = "a_short_string";
char* long_string = "a_long_long_long_long_long_long_long_string";

\.LC0:
    .string    "a_short_string"           // an unaligned string
    ...
    .align 32
\.LC1:                                    // aligned to cache line
    .string    "a_long_long_long_long_long_long_long_string"
```

Figure 4. GCC x86 string disassembly

What is necessary here is to align both strings to long words with .align 4. This uses less space and has better alignment. On a typical Linux build, this saves about 8K.

6. Cache Alignment of Structures

Arrays and lists of structures offer an opportunity to cache align large amounts of data. If the frequently accessed fields are collected into a single cache line, they can be loaded with a single memory access. This can reduce latency and cache footprint. However, it can also increase cache footprint if large amounts of data are being accessed. In this case, packing efficiency and also cache pollution are more important.

So for arrays, the base of an array should be cache aligned. The size of a structure must be either an integer multiple or an integer divisor of the cache line size. If these conditions hold, then by induction, each element of the array the cache line will be aligned or packed. Linked structures are analogous for alignment but don't have the size constraint.

An array of structures of type mem_map_t is used by the page allocator as a software page table:

```
/*
 * Try to keep the most commonly accessed fields in single cache lines
 * here (16 bytes or greater).  This ordering should be particularly
 * beneficial on 32-bit processors. ...
 */
typedef struct page {                               // from linux-2.4.0-test2
    struct list_head        list;           // 2,4
    struct address_space*   mapping;        // 1,2
    unsigned long           index;          // 1,2
    struct page*            next_hash;      // 1,2
    atomic_t                count;          // 1,1+1
    unsigned long           flags;          // 1,2
    struct list_head        lru;            // 2,4
    wait_queue_head_t       wait;           // 5,10
    struct page**           pprev_hash;     // 1,2
    struct buffer_head*     buffers;        // 1,2
    unsigned long           virtual;        // 1,2
    struct zone_struct*     zone;           // 1,2
} mem_map_t;                                // 18 * 4 ==  72 x86
                                            // 36 * 4 == 144 Alpha
```

Figure 5. mem_map_t from <linux/mm.h>

On a 32-bit Pentium, the size of mem_map_t is 72 bytes. It was 40 bytes in 2.2.16. Since the array allocation code uses sizeof(mem_map_t) to align the array, the base is aligned incorrectly as well. In any case MAP_ALIGN() can be replaced with L1_CACHE_ALIGN() which uses simpler code:

```
#define MAP_ALIGN(x) ((((x) % sizeof(mem_map_t)) == 0)            \
    ? (x) : ((x) + sizeof(mem_map_t) - ((x) % sizeof(mem_map_t))))

lmem_map = (struct page *)(PAGE_OFFSET +
   MAP_ALIGN((unsigned long)lmem_map - PAGE_OFFSET));

#define L1_CACHE_ALIGN(x) (((x)+(L1_CACHE_BYTES-1))                \
    &~(L1_CACHE_BYTES-1))

lmem_map = (struct page*) L1_CACHE_ALIGN((unsigned long) lmem_map);
```

Figure 6. lmem_map alignment

On a 64-bit Alpha, a long is 8 bytes with an 8 byte alignment and sizeof(mem_map_t) is 144 bytes. The flags field doesn't need to be a long, it should be an int. Since atomic_t is also an int and the two fields are adjacent, they would pack into a single long word. The page wait queue head used to be a pointer. Changing it back would save enough to allow cache aligning both 32-bit and 64-bit versions.

7. Cache Line Alignment for Different Architectures

It is possible to target and conditionally compile for a particular processor. Linux has an include file, <asm-i386/cache.h>, defining the L1 cache line size, L1_CACHE_BYTES, for the x86 architecture family. The slab allocator [Bonwick94], which allocates small objects from memory pages, uses L1_CACHE_BYTES when a client requests a cache aligned object with the SLAB_HWCACHE_ALIGN flag.

```
/*
 * include/asm-i386/cache.h
 */
#ifndef __ARCH_I386_CACHE_H
#define __ARCH_I386_CACHE_H
/* bytes per L1 cache line */
#if    CPU==586 || CPU==686
#define        L1_CACHE_BYTES  32
#else
#define        L1_CACHE_BYTES  16
#endif
#endif
```

Figure 7. <asm-i386/cache.h>

If someone got a Red Hat kernel conservatively compiled targeting the 486, then it assumed 16 byte cache lines. It was also wrong for the Athlon. This has been fixed in 2.4 by defining and using the kernel configuration macro CONFIG_X86_L1_CACHE_BYTES in <linux/autoconf.h>.

If you must assume one cache line size when laying out the fields inside of structs intended for portable software, use 32 byte cache lines. For example, mem_map_t could use this. Notice that 32 byte aligned cache lines are also 16 byte aligned. The PowerPC 601 nominally has a 64 byte cache line but it really has two connected 32 byte cache lines. The Sparc64 has a 32 byte L1 and a 64 byte L2 cache line. It is much easier to think of all systems as having 32 byte cache lines and enumerate the exceptions, if any. Alpha and Sparc64 have 32 byte cache lines but the Athlon and Itanium, the exceptions that proves the rule, have 64 byte cache lines. And the IBM S/390 G6 has a 256K L1 cache with 128 byte cache lines.

On the vast majority of processors, 32 byte cache lines is the right thing to do. And most importantly, if you have addressed and avoided the footprint and worst case thrashing scenarios in the 32 byte case, you will have avoided them for the other cases.

8. Caching and the Linux Scheduler

Linux represents each process with a task_struct which is allocated two 4K pages. The task list is a list of the task_struct's of all existing processes. The runqueue is a list of the task_struct's of all runnable processes. Each time the scheduler needs to find another process to run, it searches the entire runqueue for the most deserving process.

Some folks at IBM [Bryant00] noticed that if there were a couple of thousand threads that scheduling took a significant percentage of the available CPU time. On a uniprocessor machine with a couple of thousand native Java threads, just the scheduler alone was taking up more than 25% of the available CPU. This gets worse on a shared memory SMP machine because memory bus contention goes up. This doesn't scale.

It turned out that the goodness() routine in the scheduler referenced several different cache lines in the task_struct. After reorganizing task_struct, goodness() now references only a single cache line and the CPU cycle count was reduced from 179 cycles to 115 cycles. This is still a lot.

Here is the important cache line, the Linux scheduling loop and the goodness() routine. The scheduler loop iterates through the entire runqueue, evaluates each process with goodness() and finds the best process to run next.

```
struct task_struct {
    ...
    long              counter;            // critical 2cd cache line
    long              priority;
    unsigned long     policy;
    struct mm_struct *mm, *active_mm;
    int               has_cpu;
    int               processor;
    struct list_head run_list;            // only first long word
    ...
};

tmp = runqueue_head.next;
while (tmp != &runqueue_head) {
    p = list_entry(tmp, struct task_struct, run_list);
    if (can_schedule(p)) {                // running on another CPU
        int weight = goodness(p, this_cpu, prev->active_mm);
        if (weight > c)
            c = weight, next = p;
    }
    tmp = tmp->next;
}

#define PROC_CHANGE_PENALTY   15          // processor affinity

static inline int goodness(struct task_struct *p,
    int this_cpu, struct mm_struct *this_mm)
{
    int weight;
    if (p->policy != SCHED_OTHER) {
        weight = 1000 + p->rt_priority; // realtime processes
        goto out;
    }
    weight = p->counter;
    if (!weight)
        goto out;                         // no quanta left
#ifdef __SMP__
    if (p->processor == this_cpu)
        weight += PROC_CHANGE_PENALTY;   // processor affinity
#endif
    if (p->mm == this_mm)                 // same thread class
        weight += 1;                      // as current?
    weight += p->priority;
out:
    return weight;
}
```

Figure 8. task_struct and scheduler loop

A long runqueue is certainly not the common case even for heavily loaded servers. This is because event driven programs essentially self schedule with poll(). The contrasting style, threading, is preferred by Java, Apache and TUX. It is ironic that poll() also had scalability problems, and on other Unix systems as well [Honeyman99]. Also, the Linux 2.4 x86 kernels increase the maximum number of threads past 4000.

On SMP machines, processes have a scheduling affinity with the last CPU they ran on. The idea is that some of the working set is still in the local cache. But the scheduler has a subtle SMP bug. When a CPU has no processes on the runqueue, the scheduler will assign it a runnable process with an affinity to another CPU. It would be wiser to first dole out more quanta to processes on the runqueue, perhaps only those with an affinity to that CPU. Even then it may be better to idle, particularly with a short runqueue.

9. Cache Line Prefetching

Modern CPUs aggressively prefetch instructions but what about data? CPUs don't prefetch data cache lines, but vectorizing compilers do and programs can. Depending on the amount of CPU processing per cache line, you may need to prefetch more than one cache line ahead. If the prefetch is scheduled sufficiently far in advance, it won't matter if the cache line is in memory rather than L2 [Intel99a].

Typically prefetching is used in multimedia kernels and matrix operations where the prefetched address can be easily calculated. Algorithms operating on data structures can use prefetch as well. The same methods apply except that the prefetched address will follow a link rather than an address calculation. Prefetching for data structures is important since memory bandwidth is increasing faster than latency is decreasing. Traversing a data structure is more likely to suffer from a latency problem. Often only a few fields in a structure are used whereas with multimedia usually every bit is examined.

9.1. Prefetching From Memory

If a prefetch instruction can be scheduled 20-25 or so instructions before the cache line will be used, the fetch can completely overlap instruction execution. The exact prefetch scheduling distance is a characteristic of the processor and memory. Superscalar processors execute more than one instruction at a time.

9.2. Prefetching From L2

If an algorithm is traversing a data structure likely to be in L2, and it can schedule a prefetch 6-10 instructions before the cache line will be used, the fetch can completely overlap instruction execution.

The Linux scheduler loop is a good candidate for cache line prefetching from L2 because goodness() is short and after the IBM patch, it only touches a single cache line.

Here is a prefetching version of the scheduler. It overlaps the prefetch of the next cache line from L2 during the execution of goodness().

```
tmp = runqueue_head.next;
while (tmp != &runqueue_head) {
    p = list_entry(tmp, struct task_struct, run_list);
    tmp = tmp->next;
    CacheLine_Prefetch(tmp->next);      // movl xx(%ebx),%eax
    if (can_schedule(p)) {
        int weight = goodness(p, this_cpu, prev->active_mm);
        if (weight > c)
            c = weight, next = p;
    }
}
```

Figure 9. Prefetching scheduler loop

By the way, moving the tmp pointer chase before the goodness() call ends up using fewer instructions than the original. And with a little effort, the loop could omit tmp as well.

```
inline void CacheLine_Prefetch(unsigned long addr)
{
    asm volatile("" : : "r" (addr));
}
```

Figure 10. CacheLine_Prefetch()

This little bit of gcc magic is actually architecture-independent assembly code. It basically means load from addr into some temporary register of the compiler's choosing. So technically, I lied. It isn't a prefetch, it's really a preload. A prefetch offers more cache control but has some restrictions.

CacheLine_Prefetch() should be specialized on different architectures to take advantage of the various prefetch instructions. In fact, CacheLine_Prefetch() should be wrapped in conditional compilation logic because it may be inappropriate on certain Early Bronze Age machines. Also, AMD uses a slightly different set of prefetch instructions not strictly based on the MMX. On the Pentium II, this could be:

```
inline void CacheLine_Prefetch(unsigned long addr)
{
    asm volatile("prefetcht0 (%0)" :: "r" (addr));
}
```

Figure 11. Pentium II CacheLine_Prefetch()

10. Caches and Iterating Through Lists

When repeatedly iterating through an array or a list of structures, be careful of cache considerations. As the number of elements increases and approaches the number of cache lines available, thrashing will gradually increase. The gradually increasing thrashing is why this performance problem is hard to find.

The Linux scheduler iterates through the runqueue to find the next process to run. Linux also uses for_each_task() to iterate through each task_struct on the task list and perform some work. Iterating through lists represents potentially large amounts of memory traffic and cache footprint. Here is the for_each_task() iterator macro:

```
#define for_each_task(p)                                          \
    for (p = &init_task ; (p = p->next_task) != &init_task ; )
```

Figure 12. for_each_task() iterator

for_each_task() can be combined with CacheLine_Prefetch(). Notice that for_each_task() uses next_task which isn't in the preferred cache line. This doubles the memory traffic and cache footprint.

```
#define for_each_task(p)                                          \
    for (p = &init_task ; p = p->next_task,                       \
        CacheLine_Prefetch(p->next_task),                         \
        p != &init_task ; )
```

Figure 13. prefetching for_each_task() iterator

As an example, when all of the processes on the runqueue have used up their scheduling quanta, Linux uses for_each_task() to dole out more:

```
recalculate:
{
    struct task_struct *p;
```

```
        spin_unlock_irq(&runqueue_lock);
        read_lock(&tasklist_lock);
        for_each_task(p)
            p->counter = (p->counter >> 1) + p->priority;
        read_unlock(&tasklist_lock);
        spin_lock_irq(&runqueue_lock);
    }
    goto repeat_schedule;
```

Figure 14. Scheduler recalculate loop

As an observation, recalculate should iterate through the runqueue instead of the task queue. It is shorter and there is no reason to dole out more quanta to sleeping processes. counter for sleeping processes will grow without bound. When the sleeping process wakes up, it may have a large amount of quanta stored up, disturbing the responsiveness of the other processes. FreeBSD [McKusick96] recomputes the scheduling priority if an awoken process was sleeping for more than a second.

Linux also uses for_each_task() occasionally when allocating a process id in get_pid():

```
for_each_task(p) {
    if(p->pid == last_pid ||
        p->pgrp == last_pid ||
        p->session == last_pid) {
        ...
    }
}
```

Figure 15. pid allocation loop

Examining the uses of for_each_task() it is possible to change the critical task_struct cache line. Some fields are moved out, some need to be added and some need to be made smaller. active_mm is an unnecessary field for purposes of the scheduler loop. The kernel type pid_t is currently a 4 byte int. However, the maximum value for a process id is PID_MAX which is 0x8000. So pid_t can be changed to an unsigned short. (NB, PID_MAX will probably increase in 2.5). priority is limited to the range 0..40 and counter is derived from it. policy is restricted to six values. processor is currently an int and NR_CPUS is 32 so changing it to an unsigned short is reasonable.

After these changes, several of the uses of for_each_task() as well as the process id hash, find_task_by_pid, restrict their references to a single cache line.

```
struct task_struct {
    ...
    unsigned char counter;              // beginning
    unsigned char priority;
    unsigned char policy_has_cpu;
        /* one char to spare */         // one
    unsigned short processor;
    unsigned short pid;                 // two
    unsigned short pgrp;
    unsigned short session;             // three
    struct mm_struct *mm;               // four
    struct task_struct *next_task;      // five
    struct task_struct *pidhash_next;   // six
    struct task_struct *run_next;       // seven
        /* one long word to spare */    // eight
    ...
};
```

Figure 16. Packed task_struct for scheduler

If possible, squeeze everything the processing loop uses into as few cache lines as possible, preferably just one. In your data structures, if you have to use short instead of int, use short. If it will make it fit, use nibbles and bits. If you can't say it in 32 bytes, perhaps you need to rephrase yourself.

11. Thrashing

OK, you might want to close your eyes about now because it's going to start getting really gory. Now we look at the worst case scenarios of thrashing. It also gets more complex.

The task_struct is allocated as two pages which are 4K aligned. The L1 cache divides the 32-bit address space among the 128 groups each with a set of 4 cache lines. If two addresses are separated by a multiple of 4K bytes, then they map to the same cache line set. So for the 4-way set associative L1 cache on a Pentium II there are 4 cache lines available for the scheduling related cache line for all of the task_structs.

Really that isn't so bad. Asking a task_struct cache line to remain in L1 for a schedule quantum is, I admit, a bit much. But the situation doesn't really improve for L2. A 256K L2 cache will provide only 64 suitable cache lines. The way this works is that a task_struct is 4K page aligned. So in the L2 cache set index, bits 5-11 will be fixed. So there are 4 bits of possibly unique cache set index, or 16 sets of 4 cache lines or 64 available cache lines.

Furthermore, L2 is managed LRU. If you are iterating through a runqueue longer than the effective L2 set depth of 64, when you exceed the set size of L2, the cache thrashes and you may as well have no cache. Then, every reschedule the scheduler is scanning the entire runqueue from memory. Prefetching is rendered useless. Also, the set is not a list of 64 cache lines but really 16 associative subsets of 4 cache lines. Thrashing begins gradually before reaching the set size of 64.

But it gets worse. I forgot to tell you about the TLB. Logical addresses are mapped to physical addresses via a two level page table in memory and an on- chip memory management unit.

```
virtual address
    page group - bits 22-31
    page address - bits 12-21
    page offset - bits 0-11

physical page lookup
    table_level_2 = table_level_1[(vaddr & mask1) >> shift1];
    page          = table_level_2[(vaddr & mask2) >> shift2];
```

Figure 17. Pentium II VM address structure and translation

These mappings are expensive to compute. The masks and shifts are free but the loads and adds are serial. The Alpha and Itanium use a 3 level page table structure. Some page tables can be so large they must be demand paged. But since the mappings rarely change, they are computed on demand and the results are cached in the TLB. The Pentium II Translation Lookaside Buffer (TLB) uses a Harvard 4-way set associative cache with 64 data entries and 32 instruction entries. The TLB replacement policy is LRU within the set. If the TLB doesn't find a match, a TLB miss takes place. The cost of a TLB miss ranges from a reported 5 cycles on a Xeon, to 60 or more cycles. On the PowerPC it is handled in software. The PowerPC TLB miss handler is 29 instructions.

Virtual memory pages are 4K. A given task_struct page will compete for 4 TLB slots. Iterating through a long linked list (64) of 4K aligned pages is really the worst case scenario for the TLB. The iteration will flush the TLB, slowly, suffering TLB misses along the way. But for the purposes of the runqueue, the TLB could just as well have no entries at all. Linux invalidates the TLB at each process context switch. Then, every entry on the runqueue will be a TLB miss regardless of the size of the runqueue. The only case where this doesn't happen is if a process context switches to itself. Even in this case, a long runqueue can thrash and flush the TLB cache. Also, the TLB miss is synchronous with the cache line miss and a TLB miss will cause the prefetch to be ignored.

Avoid iterating through collections of 4K pages. It is a worst case cache and TLB scenario. Actually it is a member of a dysfunctional family of worst case scenarios. If addresses differ by a power of the cache line size, 64, 128, 256, ..., they compete for cache lines. Each increase in the exponent halves the number of cache lines available, down to the minimum of the cache set size. If addresses differ by a power of the page size, 8192, 16384, ..., they compete for diminishing TLB slots. And while it is possible to add more L2 on some systems, this isn't possible for TLB slots.

12. Pollution

An application can use prefetching to overlap execution with memory fetch. But once referenced, some memory is used only once and shouldn't evict more important cache lines and shouldn't take up space in the cache. If prefetch determines what cache lines will be needed, the cache control instructions determine what cache lines will be retained. With respect to performance, avoiding cache pollution is as important as prefetch.

As an example, there is a problem with the cache line oriented task_struct. The for_each_task() macro iterates through every task, polluting the L2 cache, flushing good data and loading junk. The junk data will mean further cache misses later.

On the Pentium II there is the non-temporal prefetchnta instruction. This loads a cache line into L1. If it is already in L2, it is loaded from L2. But if it isn't in L2, it isn't loaded into L2. The PowerPC doesn't have this sort of instruction. On that processor, prefetch first and then cache flush a junk cache line after using it. A junk cache line in this case would be a process not on the runqueue. This reduces L2 cache pollution but doesn't avoid it altogether as in the Pentium II prefetchnta example.

```
inline void CacheLine_Flush(unsigned long addr)
{ /* no cache line flush on the x86 */ }

inline void CacheLine_NT_Prefetch(unsigned long addr)
{
    asm volatile("prefetchnta (%0)" :: "r" (addr));
}

#define for_each_task(p)                                          \
    for (p = &init_task; p->next ? 1 : CacheLine_Flush(p),        \
        p = p->next_task, CacheLine_NT_Prefetch(p->next_task),    \
        p != &init_task;)
```

Figure 18. Pollution avoiding prefetching

Interrupt handlers and basic functions such as copying i/o buffer cache pages to and from user space with memcpy() would benefit from non-temporal prefetch, cache flush and streaming store instructions to avoid polluting L2. The Linux 2.4 x86 _mmx_memcpy() prefetches source cache lines. For a Pentium II, it should use prefetchnta for both the source and destination in order to avoid flushing and polluting L2.

On the Pentium II, an initial preload is necessary to prime the TLB for any following prefetch instructions. Otherwise the prefetch will be ignored. This is another argument against iterating through lists or arrays of 4K structures: a preload is necessary to prime the TLB but the preload will pollute the cache and there is no cache line flush instruction on the Pentium II.

As an example, this is a Pentium II user mode cache line block copy for up to a 4K page. This function assumes that the source and destination will not be immediately reused. The prefetchnta instructions marks the source and destination cache lines as non temporal. A Pentium III version [Intel99b] can use movntq and the streaming instructions. However, the streaming instructions require additional OS support for saving the 8 128-bit Katmai registers at context switch. Patches are available for 2.2.14+ [Ingo00].

```
void PII_BlockCopy(char* src, char* dst, int count)
{
    char    *limit;

    asm volatile("" :: "r" (*src));  // prime the TLB for prefetch
    asm volatile("" :: "r" (*dst));
    asm volatile("" :: "r" (*(src + 4095)));  // src may span page
    asm volatile("" :: "r" (*(dst + 4095)));

    for (limit = src + count; src < limit; src += 32, dst += 32) {
        asm volatile("prefetchnta (%0)"  :: "r" (src));
        asm volatile("prefetchnta (%0)"  :: "r" (dst));
        asm volatile("movq 00(%0),%%mm0" :: "r" (src));
        asm volatile("movq 08(%0),%%mm1" :: "r" (src));
        asm volatile("movq 16(%0),%%mm2" :: "r" (src));
        asm volatile("movq 24(%0),%%mm3" :: "r" (src));
        asm volatile("movq %%mm0,00(%0)" :: "r" (dst));
        asm volatile("movq %%mm1,08(%0)" :: "r" (dst));
        asm volatile("movq %%mm2,16(%0)" :: "r" (dst));
        asm volatile("movq %%mm3,24(%0)" :: "r" (dst));
    }
    asm volatile("emms");           // empty the MMX state
}
```

Figure 19. PII_BlockCopy()

Another candidate is memset(). Idle task page clearing with memset() has been tried but it isn't done because of cache pollution. Memory stores fill up cache lines just as memory loads do. But cache pollution can be avoided by prefetchnta of the destination cache line followed by the store. prefetchnta a priori tags the destination cache line as non cacheable. Another alternative on the PowerPC is marking the page to be cleared as non-cacheable but this is privileged [Dougan99].

13. False Sharing

A variation on the theme of thrashing is False Sharing [HennPatt96]. Two variables contained in a single cache line are updated by two different CPUs on a multiprocessor. When the first CPU stores into its variable in its cache line copy, it invalidates the cache line copy in the second CPU. When the second CPU stores into its variable, reloads the cache line, stores into it and invalidates the cache line copy in the first CPU. This is thrashing. Allocating each variable its own cache line solves this problem.

An example from the scheduler, showing the problem and solution, is the current task array variable:

```
struct task_struct *current_set[NR_CPUS];    // Linux 2.0.35

static union {                               // Linux 2.2.14
    struct schedule_data {
        struct task_struct * curr;
        cycles_t last_schedule;
    } schedule_data;
    char __pad [SMP_CACHE_BYTES];
} aligned_data [NR_CPUS] __cacheline_aligned = { {{&init_task,0}}};
```

Figure 20. SMP schduler global data array

14. Page Coloring

Two memory lines separated by modulo 64K compete for just 4 L2 cache lines. Within this 64K span, memory lines do not compete. Breaking this 64K up into 16 4K memory pages, each has a unique page color. Physical memory pages of different color don't compete for cache.

Ideally if two virtual memory pages will be used at the same time, they should be mapped to physical pages of different color [Lynch93]. In particular, simple direct mapped caches only have a single suitable cache line. Pages 0,1,2,3 are in contiguous order and of different page color. Pages 16,1,18,35 also each have different color. Page coloring also improves performance repeatability.

Higher levels of L2 associativity argue against the expense of supporting page coloring in the OS. Linux does not currently support page colored allocation because it would be too expensive. FreeBSD attempts to allocate pages in color order and there has been discussion of this for Linux if an inexpensive page allocation approach can be found.

However, page coloring is supported in one special case: __get_dma_pages(). This kernel function allocates up to 32 physically contiguous pages. Therefore pages in color order. A hint flag for the mmap() system call could request this.

15. Constraints

Memory lines within aligned structures, perhaps aligned within pages, are constrained. The greater the alignment constraint, the fewer eligible cache lines. Constraints cause conflict cache misses.

For example, in the task_struct case the important cache line is competing for 64 entries. This is known as a hot cache line: it has a constraint problem. It would be better for scheduling purposes to prune off the scheduling information and set up a slab allocated cache. The task_struct already has several data structures hanging off of it to support sharing data structures among related threads. This would be one more.

A slab is a page from which objects are allocated and freed. If a slab is full, another is constructed and the object is allocated from it. Iterating through dynamic structures allocated from slabs suffers fewer TLB misses because the structures are packed into pages.

For larger structures, frequently accessed fields are often clumped together, usually at the beginning of the structure, causing a constraint problem. Since slabs are page aligned, the slab allocator balances the cache load transparent to its clients. An offset which is a multiple of the cache line size is added as a bias.

16. Back To The Future

One solution for the scheduler is fairly simple: for an aligned array of structures, if the size of the structure is an odd multiple of the cache line size, it won't have a constraint problem.

Here is a single cache line version (one is an odd number) of the critical scheduling fields:

```
struct proc {
    unsigned char        priority;
    unsigned char        policy_has_cpu;
    unsigned short       processor;        // one
    unsigned short       pid;
    unsigned short       pgrp;             // two
    unsigned short       session;          // three - spare short
    struct mm_struct*    mm;               // four
    struct proc*         next_task;        // five
    struct proc*         pidhash_next;     // six
    struct proc*         run_next;         // seven
    struct task_struct*  task_struct;      // eight
};
```

Figure 21. Single cache line task_struct

Or it can be made into a two cache line structure and a few other task_struct fields can be added. If you are familiar with old Unix implementations, the cache line oriented task_struct is the reinvention of the proc structure. Quoting the 1977 Unix Version 6 <unix/proc.h> header file:

```
/*
 * One structure allocated per active
 * process.  It contains all data needed
 * about the process while the
 * process may be swapped out.
 * Other per process data (user.h)
 * is swapped with the process.
 */
```

Figure 22. <unix/proc.h> from 1977 Unix Version 6

The Linux scheduler searches the entire runqueue each reschedule. The FreeBSD runqueue is simpler [McKusick96]. It was derived from the 1978 VAX/VMS rescheduling interrupt handler which was all of 28 instructions. From the FreeBSD <kern/kern_switch.h> source file:

```
/*
 * We have NQS (32) run queues per scheduling class.  For the
 * normal class, there are 128 priorities scaled onto these
 * 32 queues.  New processes are added to the last entry in each
 * queue, and processes are selected for running by taking them
 * from the head and maintaining a simple FIFO arrangement.
 * Realtime and Idle priority processes have and explicit 0-31
 * priority which maps directly onto their class queue index.
 * When a queue has something in it, the corresponding bit is
 * set in the queuebits variable, allowing a single read to
 * determine the state of all 32 queues and then a ffs() to find
 * the first busy queue.
 */
```

Figure 23. From FreeBSD <kern/kern_switch.h>

A uniprocessor reschedule is simplicity:

```
static struct rq    queues[32];
static u_int32_t    queuebits;
int                 qi;

qi = (current->priority >> 2);
SetBit(&queuebits, qi);
InsertQueue(&queues[current->priority], current);

qi = FindFirstSet(queuebits);
if (RemoveQueue(&queues[qi], &current) == Q_EMPTY)
    ClearBit(&queuebits, qi);
```

Figure 24. Queueing scheduler context switch

A best of breed Linux scheduler would use the FreeBSD runqueue and support the Linux scheduler policies of the goodness() function: soft realtime scheduling, SMP CPU affinity and thread batching.

17. In Summary, You Are Doomed

Cache programming is a collection of arcana and techniques. What follows is an attempt to organize them and the final chart is an attempt to distill them.

17.1. Alignment and Packing

When laying out structures for portable software, assume a 32 byte cache line.

For arrays, the base of an array should be cache aligned. The size of a structure must be either an integer multiple or an integer divisor of the cache line size. If these conditions hold, then by induction each element of the array the cache line will be aligned or packed. Linked structures are analogous for alignment but don't have the size constraint.

To specify a cache line alignment for types or variables with gcc, use the aligned attribute:

```
struct Box {
    int    element;
} __attribute__ ((aligned(SMP_CACHE_BYTES)));
```

Figure 25. Cache alignment of structure types

In an application, to allocate cache aligned memory, use memalign(32, size) instead of malloc(size).

Data structures change. Write a program which validates packing, alignments and worst case scenarios.

17.2. Cache Footprint

Reduce your cache footprint. Paraphrasing the Elements of Style [Strunk99]:

- Omit needless words.
- Keep related words together.
- Do not break cache lines in two.
- Avoid a succession of loose cache lines.
- Make the cache line the unit of composition.

When repeatedly iterating through a large array or list of structures, be careful of cache considerations. As the number of elements increases and approaches the number of cache lines available, cache misses will gradually increase to the point of thrashing.

Ignoring page color issues for now, large memory scans approaching the size of the cache, flush the cache. Large memory copies approaching half of the cache, flush the cache. Both pollute the cache with junk.

Avoid polluting the cache with junk data. If a cache line won't be reused for some time, flush it. If streaming through a lot of data, non-temporally prefetch source and destination to avoid polluting the L2 cache. Otherwise you may pollute the cache with junk data entailing further cache misses.

Interrupt handlers should avoid cache pollution by using non-temporal prefetch, cache flush and streaming store instructions.

Shared libraries allow many processes to share the same instruction memory. This creates more opportunities for cache sharing and less cache competition.

17.3. Prefetch

If you are working on one cache line and then another, prefetch the next cache line before doing the work on the first. This overlaps the work and the prefetch. Prefetch hides memory latency.

The prefetch scheduling distance for memory is about 20-25 instructions. The prefetch scheduling distance for L2 is about 6-10 instructions. These are very rough guides.

17.4. Worst Case Scenarios

Avoid worst case scenarios: for associative cache memory, if addresses differ by a power of the cache line size, 64, 128, 256, ..., they are constrained to use fewer cache lines. Each increase in the exponent halves the number of cache lines available, down to the minimum of the cache set size. Similarly, on an associative TLB, if addresses differ by a power of the page size, 8192, 16384, ..., they are constrained to use fewer TLB slots with each increase in exponent.

False Sharing - avoid updating a cache line shared between two or more CPUs.

Constraints - bias the allocation of small objects by a multiple cache line offset to avoid hot cache lines. The slab allocator does this. For an array of structures, use odd multiples of the cache line size.

Coloring - if necessary, allocate physical memory pages in color order. They won't compete for cache.

If you need uniquely colored pages in the kernel or in a module, use __get_dma_pages(gfp_mask, order) to allocate up to 32 pages, usually 128K. They will be physically contiguous and therefore in color order.

17.5. General

Cache Policy	Cache Miss	Description	Suggestion
load	compulsory	first access	align, pack and prefetch
placement	conflict	access mismatched to cache design	avoid worst cases
replacement	capacity	working set is larger than cache size	avoid cache pollution
store	combine	single store or large streaming store	all of the above, combine writes, use non temporal instructions

18. Acknowledgements

Scott Maxwell, David Sifry and Dan Sears suffered through and improved earlier drafts.

19. References

[Bonwick94] Jeff Bonwick, The Slab Allocator: An Object-Caching Kernel Memory Allocator, *Usenix Summer 1994 Technical Conference*, 1994.

[Bryant00] Ray Bryant and Bill Hartner, Java technology, threads, and scheduling in Linux: Patching the kernel scheduler for better Java performance, *IBM Developer Works*, January 2000.

[Dougan99] Cort Dougan, Paul Mackerras and Victor Yodaiken, Optimizing the Idle Task and Other MMU Tricks, *Third Symposium on Operating Systems Design and Implementation*, 1999.

[Handy98] Jim Handy, The Cache Memory Book, 2nd edition, Academic Press, 1998.

[HennPatt96] John Hennessy and David Patterson, Computer Architecture, 2nd edition, Morgan Kauffman, 1996.

[Honeyman99] Peter Honeyman et al, The Linux Scalability Project, University of Michigan CITI-99-4, 1999.

[Ingo00] Ingo Mulnar, *http://people.redhat.com/mingo/mmx-patches/*

[Intel99a] Intel Architecture Optimization Reference Manual, *developer.intel.com*, 1999.

[Intel99b] Block Copy Using Intel Pentium III Processor Streaming SIMD Extensions, *developer.intel.com*, 1999.

[Lynch93] William L. Lynch, The Interaction of Virtual Memory and Cache Memory, Stanford CSL-TR-93-587, 1993.

[Maxwell99] Scott Maxwell, Linux Core Kernel Commentary, Coriolis, 1999.

[McKusick96] Kirk McKusick et al, The Design and Implementation of the 4.4BSD Operating System, Addison Wesley, 1996.

[Schimmel94] Curt Schimmel, UNIX Systems for Modern Architectures: Symmetric Multiprocesssing and Caching for Kernel Programmers, Addison Wesley, 1994.

[Seward] Julian Seward, Cacheprof, *www.cacheprof.org*

[Shanley97] Tom Shanley, Pentium Pro and Pentium II System Architecture: 2nd edition, Mindshare, 1997.

[Stallman00] Richard Stallman, Using and Porting GNU CC: for version 2.95, Free Software Foundation, 2000.

[Strunk99] Strunk and White, The Elements of Style, 4th edition, Allyn and Bacon, 1999.

A PPPoE Implementation for Linux

David F. Skoll
Roaring Penguin Software Inc.
dfs@roaringpenguin.com, http://www.roaringpenguin.com

Abstract

Many DSL service providers use PPPoE for residential broadband Internet access. This paper briefly describes the PPPoE protocol, presents strategies for implementing it under Linux and describes in detail a user-space implementation of a PPPoE client.

1 Introduction

Many Internet service providers are using the Point-to-Point Protocol over Ethernet (PPPoE) to provide residential Digital Subscriber Link (DSL) broadband Internet access. Most ISP's do not support Linux and supply PPPoE clients only for Windows and Mac OS. This paper describes a PPPoE client for Linux.

The paper is organized as follows: Section 2 provides an introduction to the PPPoE protocol and a brief discussion of why it is used. Section 3 describes the various strategies which can be used to implement PPPoE under Linux. Section 4 describes `rp-pppoe`, a particular user-space implementation of PPPoE. Section 5 describes additional PPPoE-related tools and ports to non-Linux systems. Finally, Section 6 contains some concluding remarks.

2 The PPPoE Protocol

PPPoE is a protocol for encapsulating PPP frames in Ethernet frames[1]. PPP is a data-link-level protocol typically used to encapsulate network-level packets over an asynchronous serial line. This mode of usage is called *asynchronous PPP*.

2.1 Asynchronous PPP

Asynchronous PPP uses *byte stuffing* to mark frame boundaries[2]. The special byte 0x7E called a *flag sequence*. The start of a frame is marked by a flag sequence followed by bytes 0xFF and 0x03. Next, a two-byte *protocol* field identifies the network-layer protocol. Next, the network layer data is sent, followed by a two- or four-byte *frame check sequence*.

To make recognition of frame boundaries unambiguous, if the flag sequence appears inside a frame, it is *escaped* by transmitting the byte 0x7D (the *escape sequence*) followed by the original byte XOR'd with 0x20. Naturally, the escape sequence itself must be escaped, and is transmitted as 0x7D, 0x5D. Other byte values may be escaped.

The receiver discards the extra escape sequences to reconstruct the original PPP frame.

2.2 Synchronous PPP

Asynchronous serial links cannot inherently mark frame boundaries, so byte-stuffing (or some equivalent) is required. For data-link types which naturally mark frame boundaries, no byte-stuffing is needed. Since Ethernet has natural frame boundaries, PPP frames can be encapsulated in Ethernet frames without any byte stuffing.

The PPPoE Session Frame consists of a PPP frame inside an Ethernet frame, with six bytes of PPPoE information. The format of this PPPoE information will be described in Section 2.4.

2.3 PPPoE Discovery Phase

A *PPPoE session* consists of two PPP peers communicating over Ethernet[3]. Each peer knows the MAC address of the other peer. In addition, a *session number* is used to uniquely identify a particular PPPoE session between two peers.

While PPP is a peer-to-peer protocol, PPPoE is initially a client-server protocol. The client (usually a personal computer) searches for a PPPoE server (called an *access concentrator*) and obtains the access concentrator's MAC address and a session number. The process of setting up a PPPoE session is called *discovery*. PPPoE discovery uses special Ethernet frames with their own Ethernet frame type (0x8863).

To initiate discovery, the PPPoE client sends a PPPoE Active Discovery Initiation (PADI) frame.

This frame is sent to the broadcast Ethernet address (FF:FF:FF:FF:FF:FF) and may specify a particular "service name" which the client is interested in.

When an access concentrator receives a PADI frame, it responds with a PPPoE Active Discovery Offer (PADO) frame, if it is willing to set up a session with the client. The destination Ethernet address of the PADO frame is the unicast Ethernet address of the client who sent the PADI.

In general, there can be more than one access concentrator within broadcast range of the client. The client therefore collects PADO responses and picks one with which it would like to start a session. It sends a PPPoE Active Discovery Request (PADR) frame to the unicast Ethernet address of the access concentrator.

If the access concentrator agrees to set up a session with the client, it allocates resources to set up a PPP session and assigns a session number. It sends this number back to the client in a PPPoE Active Discovery Session-confirmation (PADS) frame. When the client receives the PADS frame, it knows the access concentrator's Ethernet address and the session number. It allocates resources to set up a PPP session.

2.4 PPPoE Session Phase

Once each side knows the other's Ethernet address and the session number, the PPP session can begin. PPP frames are encapsulated in PPPoE session frames, which have Ethernet frame type 0x8864. A PPPoE session frame is shown in Figure 1.

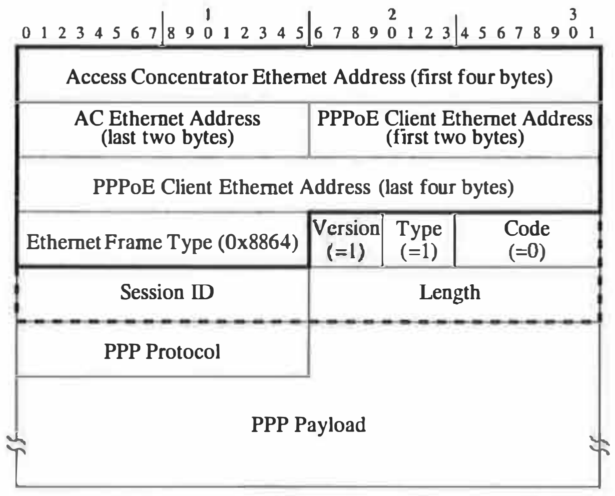

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
Access Concentrator Ethernet Address (first four bytes)
AC Ethernet Address (last two bytes)
PPPoE Client Ethernet Address (first two bytes)
PPPoE Client Ethernet Address (last four bytes)
Ethernet Frame Type (0x8864)
Version (=1)
Type (=1)
Code (=0)
Session ID
Length
PPP Protocol
PPP Payload

Figure 1: PPPoE Session Frame

In the session frame, the four-bit fields **Version** and **Type** are set to 1. The **Code** field is used in the discovery phase to identify the packet type, but is always set to zero in the session phase. The **Session ID** field is the session ID assigned during discovery. The **Length** field is the length of the PPP payload, not including the Ethernet or PPPoE headers.

The PPP data begins with the PPP protocol field. Note that no flag sequences are included. The PPP data is not byte-stuffed with the escape sequence, and does not include the final PPP frame-check sequence. (The FCS is omitted because Ethernet frames have their own frame check sequence, and there is no point in duplicating it.)

2.5 Why PPPoE?

PPPoE has many advantages for DSL service providers, and practically none for DSL consumers.

- Because PPPoE sessions are really just PPP sessions, IP addresses can be very dynamic. There is no possibility to hold on to a fixed IP addresses by renewing a DHCP lease frequently. Service providers can ensure that your assigned IP address is changed each time you connect.

- Because PPPoE creates the concept of a "session" over Ethernet, service providers can charge based on connect time. This allows them to discourage permanent connections and over-subscribe their IP address pool.

- Because PPP sessions almost always require authentication, DSL service providers can bill the correct client regardless of where he connects from (as dial-up ISP's can now.)

In theory, PPPoE offers the following advantages to end users. In practice, these advantages are either negligible or not implemented by the service provider.

- PPPoE can encapsulate non-IP protocols. Any protocol which can be encapsulated by PPP can be sent via PPPoE.

- Service providers can enter into agreements with large organizations to authenticate users and provide dedicated sessions behind the organizations' firewall (for employees who need remote access, for example.)

No ISP that I'm aware of supports non-IP protocols over PPPoE, and access through a firewall is better achieved with SSH or IPSec.

3 Implementing PPPoE under Linux

There are many possibilities for implementing a PPPoE client under Linux. The following are the three reasonable strategies:

1. Implement both the PPPoE discovery and session phases in a user-space program.

2. Implement PPPoE discovery in a user-space program and PPPoE session in the kernel.

3. Implement both the PPPoE discovery and session phases in the kernel.

We can dispose of the third strategy right away. PPPoE discovery is not speed-critical and consists of code which is seldom used. There's no point in bloating the kernel with discovery code.

The choice then boils down to including PPPoE session code in the kernel or in a user-space program. Again, the choice is clear: The kernel itself should handle the PPPoE session. Executing user code for each PPPoE frame is very inefficient.

As of this writing, PPPoE support exists in the experimental 2.3 kernels and is expected to be included in the final 2.4 kernels. In addition, there are kernel patches to add PPPoE support to 2.0 and 2.2 kernels.

Although the kernel is clearly the right place for PPPoE session support, I have implemented PPPoE in a user-space program. While this may be repugnant to kernel hackers, there are some advantages to a user-space program:

- A user-space program is easy to write and debug.
- A user-space program is relatively portable. In fact, `rp-pppoe` has already been ported to NetBSD (by Geoff Mottram and Yannis Sismanis.)
- Many Linux users are novices, and the existing kernel patches are not easy for them to apply and configure. A simple RPM or DEB package with a helpful configuration shell script is much more palatable for new users.
- Many Linux users do not want to modify their kernels. If they ever need to upgrade the kernel for security reasons, they do not want to remember to have to compile in PPPoE support. (This reason will disappear once standard kernels include PPPoE.)
- Most residential DSL connections are slow (2.2Mb/s or less) and even a user-space client can keep up with this on all but the oldest hardware.

I therefore view the user-space client as a convenient but temporary measure until stable kernels include PPPoE support. Even after the standard kernel supports PPPoE, most of the user-space code involves the discovery phase, and can be re-used in newer kernels. The session code is very simple and there's no great loss in discarding it.

4 The `rp-pppoe` User-Space PPPoE Client

`rp-pppoe` is a free (under the GNU General Public License) user-space PPPoE implementation for Linux. Figure 2 illustrates the operation of `rp-pppoe`.

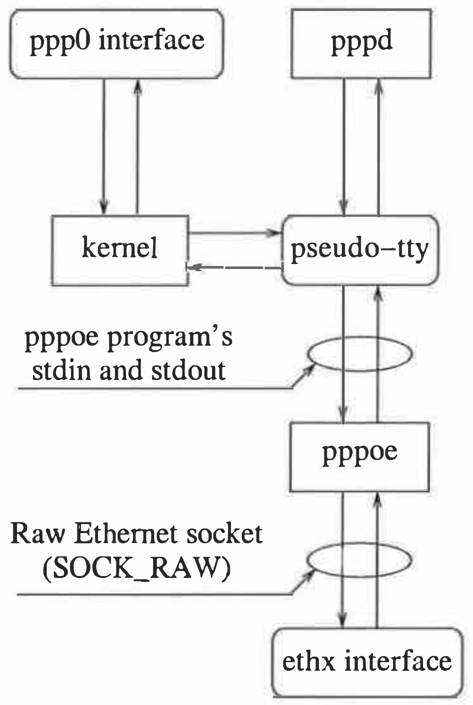

ppp0 interface
pppd
kernel
pseudo–tty
pppoe program's stdin and stdout
pppoe
Raw Ethernet socket (SOCK_RAW)
ethx interface

Figure 2: `rp-pppoe` Configuration

4.1 The Pseudo-TTY

The `pppd` program and the Linux kernel expect to transmit PPP frames over a TTY device. Luckily, UNIX (and Linux) support the concept of a *pseudo-tty*. This is a device which "looks" like a TTY, but instead of being connected to a physical terminal, it is connected to a UNIX process. Whenever something writes to the pseudo-tty, the data appears on the standard input of the back-end process. Whenever the back-end process writes to its standard output, the data may be read from the pseudo-tty.

Even more luckily, recent versions of `pppd` (2.3.7 and newer) support a `pty` option. This option automatically starts the back-end process and performs all the mundane operations required to connect it to a pseudo-tty.

So to start the PPPoE link, you start `pppd` with the appropriate `pty` option, which runs the `pppoe` executable connected to the pseudo-tty.

4.2 The Discovery Phase

Once `pppoe` begins executing, it starts PPPoE discovery. It creates a raw Ethernet socket. This special socket allows user-space programs to transmit and receive raw Ethernet frames.

`pppoe` constructs and transmits a PADI frame, and waits for PADO frames. When a PADO frame arrives (if it meets criteria specified on the `pppoe` command line), `pppoe` transmits a PADR frame. Once it receives a PADS frame, it records the session ID and moves to the session phase.

Note that Linux raw sockets perform only limited filtering on Ethernet frames. They are not nearly as flexible as the Berkeley Packet Filter found on BSD systems. Luckily, for PPPoE, filtering only on the Ethernet frame type is acceptable, and Linux raw sockets can perform this level of filtering. (We want to filter out non-PPPoE frames in the kernel; otherwise, non-PPPoE traffic could consume huge amounts of CPU time as pppoe is scheduled in to read the frame.)

4.3 The Session Phase

Once the session phase begins, pppoe reads asynchronously-framed PPP data on standard input, and writes it to standard output. Let's trace these operations.

When a frame is transmitted over the PPP interface, pppoe's standard input becomes readable. pppoe reads from standard input and collects data until it has assembled an entire PPP frame.

Note that pppoe must keep a small state machine to record where it is in the PPP frame assembly. There's no guarantee that it will read an entire PPP frame in one chunk, or that it won't read more than one frame. This is because UNIX write operations do not preserve write boundaries; if you write one byte to a pseudo-tty followed by three bytes, the back-end process may read all four bytes at once, depending on scheduling.

During PPP frame assembly, pppoe removes escape sequences and "de-stuffs" the frame. This converts the asynchronous PPP framing into a synchronous PPP frame.

Once the PPP frame is assembled, PPPoE headers are added and the frame is transmitted over the raw socket. (Actually, most of the PPPoE headers are constant for a given session, so the PPP frame is simply assembled into a buffer right after the constant PPPoE header portions.)

When an incoming PPPoE frame is received by the Ethernet card, pppoe's raw socket becomes readable. In this case, we are guaranteed that a read operation will return one (and only one) frame, and will return the entire frame if the buffer is big enough. pppoe reads the frame into a buffer. It then adds asynchronous byte-stuffing to the data and computes the PPP frame-check sequence. (Recall that the PPP FCS is not transmitted over PPPoE.) Finally, it writes the result to standard-output, where it is picked up by the kernel or pppd.

4.4 Synchronous PPP

You can immediately see two gross inefficiencies: User-space code is executed for every PPPoE frame, and byte-stuffing and de-stuffing is done twice. For outgoing frames, the kernel carefully performs byte stuffing, which is undone by pppoe. For incoming frames, pppoe stuffs them and the kernel de-stuffs them.

There is an option to pppd and pppoe which enables synchronous PPP. In this case, no byte-stuffing is performed. However, correct operation in this mode relies on pppoe reading exactly one frame at a time from standard input. (There are no frame boundary markers, so pppoe assumes that it gets a complete frame for each read system call.) While this seems to work on fast machines, it is not recommended, because delays in scheduling in pppoe can cause serious problems.

The kernel-mode implementation of PPPoE has no problem guaranteeing frame boundaries (because kernel code is invoked for each read and write call), so the kernel-mode implementation uses synchronous PPP without worries.

4.5 The MTU

PPPoE introduces a real and annoying problem. The maximum Ethernet frame is 1518 bytes long. 14 bytes are consumed by the header, and 4 by the frame-check sequence, leaving 1500 bytes for the payload. For this reason, the Maximum Transmission Unit (MTU) of an Ethernet interface is usually 1500 bytes. This is the largest IP datagram which can be transmitted over the interface without fragmentation.

PPPoE adds another six bytes of overhead, and the PPP protocol field consumes two bytes, leaving 1492 bytes for the IP datagram. The MTU of PPPoE interfaces is therefore 1492 bytes.

When a TCP connection is initiated, each side can optionally specify the Maximum Segment Size (MSS). TCP chops a stream of data into segments, and MSS specifies the largest segment each side will accept. By default, the MSS is chosen as the MTU of the outgoing interface minus the usual size of the TCP and IP headers (40 bytes), which results in an MSS of 1460 bytes for an Ethernet interface.

TCP stacks try to avoid fragmentation, so they use an MSS which will not cause fragmentation on their outgoing interface. Unfortunately, there may be intermediate links with lower MTU's which will cause fragmentation. Good TCP stacks perform *path MTU discovery*.

In path MTU discovery, a TCP stack sets a special Don't Fragment (DF) bit in the IP datagrams. Routers which cannot forward the datagram without fragmenting it are supposed to drop it and send an ICMP "Fragmentation-Required" datagram to the originating host. The originating host then tries a lower MTU value.

Unfortunately, many routers are anti-social and do not generate the fragmentation-required datagrams. Many firewalls are equally anti-social and drop all ICMP datagrams.

Now consider a client workstation on an Ethernet LAN connected to a PPPoE gateway. It opens a TCP connection to a web server. Because the Ethernet MTU is 1500, it suggests an MSS of 1460. The web server is also on an Ethernet and also suggests an MSS of 1460. The client then requests a web page. This request is typically small and reaches the web server. The server responds with many TCP segments, most of which are 1460 bytes long.

The maximum-sized segments result in 1500-byte IP datagrams and make their way to the DSL provider. The DSL provider cannot transmit a 1500-byte IP datagram over a PPPoE link, so it drops it (assume for now that the DF bit is set.) Furthermore, being anti-social, the DSL provider does not send an ICMP message to the web server.

The net result is that packets are silently dropped. The web client hangs waiting for data, and the web server keeps retransmitting until it finally gives up, or the connection is closed by the user aborting the web client.

One way around this is to artificially set an MSS for the default route on all LAN hosts behind the PPPoE gateway. This is annoying, as it requires changes on each host.

Instead, `rp-pppoe` "listens in" on the MSS negotiation and modifies the MSS if it is too big. (This was inspired by `mssclampfw` by Marc Boucher.)

`rp-pppoe` can be configured to intercept all TCP packets with the SYN bit set and silently adjust any advertised MSS options so they will be appropriate for the PPPoE link. Although the MSS option can appear in any TCP packet, in practice, most implementations send it only with SYN and SYN-ACK packets.

Adjusting the MSS is a gross hack. It breaks the concept of the transport-layer being end-to-end. It will not work with IPSec, because IPSec will not let you damage IP packets (they will fail to authenticate.) Nevertheless, it is a fairly effective solution to an ugly real-world problem, and is used by default in `rp-pppoe`.

5 Additional PPPoE Tools and Ports

In addition to the PPPoE client, the `rp-pppoe` package includes a couple of other useful tools: `pppoe-server` implements a PPPoE server, and `pppoe-sniff` examines frames from non-Linux systems to determine if any special run-time options are required to establish a connection.

5.1 The PPPoE Server

The PPPoE server is a very simple program. It listens for PPPoE discovery frames. When a PADI frame is received, it constructs a cookie by hashing the peer Ethernet address, the server's Ethernet address and a random number. This cookie is sent back in a PADO frame.

The cookie is designed to stop a simple-minded denial-of-service attack. In this attack, a malicious client sends many PADR frames with fake source Ethernet addresses. If the server responded to the PADR with a PADS, it would have to allocate resources for a PPP connection. Flooding the server with many PADR frames could quickly exhaust its resources. For this reason, requiring the client to return the cookie in its PADR frame ensures that the PADI was received from a valid Ethernet address.

The server can take further measures to limit denial-of-service attacks (such as limiting the number of PPP sessions per Ethernet address), but the current implementation of `pppoe-server` does not do that.

Once `pppoe-server` has received a valid PADR frame, it responds with a PADS and simply forks and execs `pppd` to handle the PPP connection. The new `pppd` process uses the `pty` option along with a special flag to `pppoe` informing it of the session number assigned by the server and the peer's Ethernet address.

Since each PPP session creates a `pppd` and `pppoe` process, `pppoe-server` is not suitable for production use as a heavy-traffic PPPoE server. It is meant to test PPPoE client ports and validate RFC-compliance of PPPoE clients.

5.2 PPPoE Sniffing

Some Internet service providers require special data in the PADI and PADR frames. For example, some providers require a `Service-Name` tag with a specific value. Unfortunately, since most providers support only Windows and Mac OS, and most ISP help-desk personnel haven't a clue about the inner workings of PPPoE, obtaining the information required to establish a PPPoE session under Linux is sometimes difficult.

The `rp-pppoe` package includes a program called `pppoe-sniff`. Figure 3 illustrates how to operate `pppoe-sniff`.

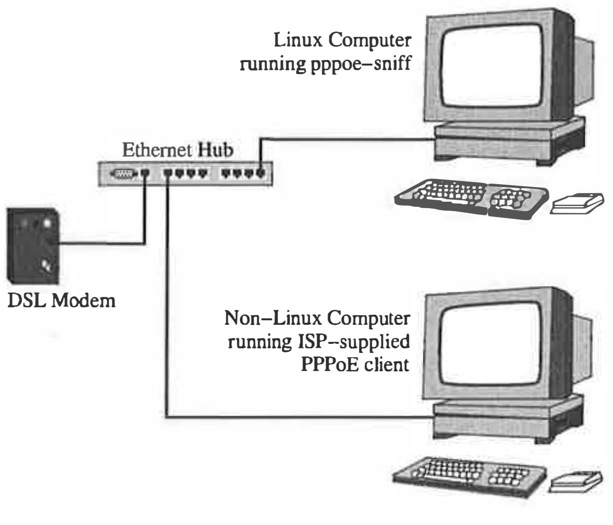

Linux Computer running pppoe–sniff
Ethernet Hub
DSL Modem
Non–Linux Computer running ISP–supplied PPPoE client

Figure 3: Using `pppoe-sniff`

Connect a Linux computer to an Ethernet hub. Connect another computer running the ISP-supported PPPoE client to the hub. Finally, connect the DSL modem either to the uplink port of the hub with a straight-through cable, or to a normal port of the hub with a crossover cable. It is important to use a hub and not a switch, because the Linux computer must be able to see all traffic between the DSL modem and the other computer. Also, the two computers and the modem should be the only devices connected to the hub, because some DSL modems react badly if they see Ethernet frames from more than two other MAC addresses.

Once the setup is complete, start `pppoe-sniff` on the Linux machine and make a connection to the ISP using the ISP-supplied client.

Because one person using `rp-pppoe` reported that his ISP used non-standard frame types for PPPoE discovery and session frames (0x3c12 and 0x3c13 instead of 0x8863 and 0x8864), `pppoe-sniff` listens for all Ethernet frames and picks out likely-looking PPPoE frames based on the type, version and code fields. It is conceivable that non-PPPoE traffic could confuse (or crash!) `pppoe-sniff`, so do not run such traffic while `pppoe-sniff` is operating.

`pppoe-sniff` looks for a likely-looking PADR frame and a likely-looking session frame. From these frames, it gleans the frame types used by the ISP, as well as any Service-Name tag which may be required during discovery. It prints its findings out in the form of command-line options to supply to `pppoe`. In this way, you can usually determine any special options required by your ISP without having to go through technical support personnel.

5.3 NetBSD Port

In addition to Linux, `rp-pppoe` has been ported to NetBSD. The port was done by Geoff Mottram and Yannis Sismanis. It involved substituting calls to NetBSD's BPF API instead of Linux's SOCK_PACKET. The port was quite easy and affected only about 10% of the total code in `rp-pppoe`. It should be easy to port `rp-pppoe` to SVR4-derived UNIXes which use DLPI, or even to write a `libpcap`- and `libnet`-based version, which will be portable to all systems to which these libraries have been ported.

(Actually, while `rp-pppoe` is quite portable, it requires the `pty` option of `pppd`, which apparently is no longer maintained on FreeBSD or OpenBSD. Anyone wanting to port `rp-pppoe` to a new UNIX will have to ensure that `pppd` is ported, too.)

6 Summary

PPPoE is not a wonderful protocol for end-users. Unfortunately, it is here to stay, and the Linux community will have to live with it.

The user-space `rp-pppoe` is a convenient method of connecting to a PPPoE provider, but it should be viewed as a temporary measure until stable kernels support PPPoE natively. Once this happens, the `rp-pppoe` software will be used only for the discovery phase, and will offload the session phase to the kernel.

Bugs in many routers prevent the correct operation of path MTU discovery. `rp-pppoe` has a gross hack to work around this for TCP connections. I would love to eliminate this hack, but it requires that all router vendors fix their software. This is not very likely to happen.

Some ISP's use non-standard frame types and/or require special Service-Name tags during the discovery phase. The `pppoe-sniff` program attempts to extract this information by listening to an ISP-supplied PPPoE client's connection.

`rp-pppoe` is available at the following URL: `http://www.roaringpenguin.com/pppoe/`. A kernel-mode PPPoE implementation is available at `http://www.davin.ottawa.on.ca/pppoe/`.

7 Acknowledgements

I'd like to thank all the people who have downloaded, tested and used `rp-pppoe`. In particular, Geoff Mottram, Yannis Sismanis, Heiko Shlittermann, Gary Cameron, Julian Gorfajn, Geoff Kuenning, Patrick Smith and Jason Lassaline submitted patches, ports and bug reports.

References

[1] RFC 1661, "The Point-to-Point Protocol (PPP)", W. Simpson, Editor, July 1994.

[2] RFC 1662, "PPP in HDLC-like Framing", W. Simpson, Editor, July 1994.

[3] RFC 2516, "A Method for Transmitting PPP Over Ethernet (PPPoE)", L. Mamakos et al., February 1999.

Linux–HA Heartbeat System Design

Alan Robertson – SuSE Labs <alanr@suse.com>

ABSTRACT

One of the most commonly identified features which is felt to be necessary for Linux™ to be considered "enterprise-ready" is High–Availability. High–Availability (HA) systems provide increased service availability through clustering techniques.

HA clusters minimize availability interruptions by quickly switching services over from failed systems to working systems, providing the customer with an illusion of continuous availability. As such, high–availability features, are vital to mission–critical systems. Although there are many components to a high–availability system, two of the key components are heartbeat services and cluster communication services. Heartbeat services provide notification of when nodes are working, and when they fail. In the Linux–HA project, the *heartbeat* program provides these services and intracluster communication services.

This paper describes the design of the *heartbeat* program which is part of the High–Availability Linux Project with particular emphasis on the rationales behind key design choices, and the results obtained.

Introduction

As Linux™[1] grows into handling larger server systems satisfactorily, it will have to provide many of the same services which these larger servers by Sun, Compaq, IBM, and others have traditionally provided. One of the key features which these larger and more mission–critical servers have provided customers is high–availability (HA) clustering.

A high–availability cluster is a group of computers which work together in such a way that the failure of any single node in the cluster will not cause the service to become unavailable. Given this definition, it seems obvious that it is necessary for the cluster to detect when servers fail, and when they become available again. This task is performed by code which is usually called "heartbeat" code. In the case of Linux–HA, this function is performed by a program called *heartbeat*. Heartbeat programs typically send packets to each machine in the cluster to indicate that they are still alive.

Another of the most basic functions which any High–Availability system must perform is cluster communications. It is often the case that these communications need to communicate between all cluster members at once in a broadcast or multicast sense.

The Linux–HA *heartbeat* program takes the approach that the keepalive messages which it sends are a specific case of the more general cluster communications service. In this sense, it treats cluster membership as joining the communication channel, and leaving the cluster communication channel as leaving the cluster. Because of this, the heartbeat messages which are its namesake are almost a side–effect of cluster communications, rather than a separate standalone facility in the *heartbeat* program. It should be emphasized that *heartbeat* should not be understood as a complete cluster management solution, but a basic component providing certain well–defined low–level services. These services are outlined in more detail below.

Heartbeat Design Philosophy

The *heartbeat* component of the Linux–HA project **[Rob00]** is in some senses a simple program. It is one of the the lowest–level components of the system, and has the purpose of being reliable, so it is important that it be simple and straightforward. It should be designed to run continuously for years without memory leaks, or bugs. It needs to be easy to understand, easy to debug, and extremely robust. For this reason, when design alternatives were considered, the simplest, most straightforward, and easiest to debug were often chosen.

Even though this low level subsystem is reasonably simple, there are some non–obvious design decisions and synergies which were made which appear to be worth understanding.

1 Linux is a trademark of Linus Torvalds.

It is the intent of this paper to explore some of these elements of the design, and talk about how it may be extended in the future.

Definitions

The following definitions will prove useful in the discussion of the design of the Linux–HA *heartbeat* program.

Heartbeat Subsystem
A subsystem which monitors the presence of nodes in the cluster through a series of kee–palive, or heartbeat messages.

Cluster Manager
A subsystem which manages the cluster, de–ciding which resources are on which nodes, and taking appropriate action during cluster transi–tions. In this document, this term is used to refer to an entire range of cluster functions and services which are direct or indirect clients of heartbeat's API.

Cluster Transition
A cluster transition occurs whenever a cluster member enters or leaves the cluster. This event triggers recovery and reconfiguration actions by the cluster management software.

Low–Level Services in High–Availability Systems

Every high–availability system needs two basic services: to be informed of cluster mem–bers joining and leaving the system, and to provide basic communication services for man–aging a cluster. As is discussed below, there is considerable synergy between these two func–tions. In the discussion below, application data is specifically excluded from consideration. All that is under discussion here is cluster control data, that is, data used for managing and con–trolling the configuration of the cluster.

Low Level Cluster Membership

The lowest level concept of cluster member–ship is typically founded on the idea of reachability: Can we communicate with node X? If so, it is considered to be a member of the cluster. If not, it is considered "dead". Towards this end, one conventionally creates a heartbeat subsystem to determine if nodes are alive or dead. In such a system, each machine which is actively a part of the cluster creates heartbeat messages on a periodic basis, and sends them out to all cluster members. Any machine whose heart can no longer be heard to beat (i.e., from whom messages are no longer heard) is consid–ered to be "dead", or have left the cluster. Such events (members leaving or joining the cluster) trigger cluster transitions.

Typically, this means sending out messages from every machine to every machine on a pe–riodic basis (perhaps once per second or so). Fo r large clusters, there is considerable advantage in using multicast or broadcast techniques to avoid the traffic associated with the $O(N^2)$ op–eration of sending messages from every machine to every machine. In a stable high–availability system (one not in the midst of a transition) these messages are virtually the only messages being sent. Indeed, these messages constitute the overwhelming majority of the communications traffic in a cluster system. Therefore, although these messages are simple, in large clusters, become quite voluminous if not handled carefully. Ultimately, this lowly heartbeat function can limit the scaleability of a high–availability system, particularly if not handled efficiently.

Note that this form of low–level (or local) cluster membership is not sufficient for a com–plete cluster management infrastructure, it needs to be combined with the idea of consen–sus or agreement wherein the members of the cluster communicate with each other and ex–change their views of cluster membership, resulting in a cluster–wide view of cluster membership. This is discussed further in the *Future Directions* section later in this paper.

Communications in High–Availability Systems

Every high–availability cluster has needs to communicate in order to operate. This commu–nication varies in content and size depending on the particular cluster management system being used in the cluster. As an example, the current heartbeat code has a message type to request that a particular resource group be relinquished, and a corresponding response message. Other resource–related messages also commonly oc–cur during cluster transitions. Although this depends greatly on what cluster management model the cluster has adopted; the majority of (non–heartbeat) cluster management messages are associated with cluster transitions.

Since cluster management is typically trans–action oriented, with the transactions spanning the entire cluster, it is common to communicate with all nodes in the cluster simultaneously. For example, one node might note an event which should trigger a cluster transition. This node would then notify all nodes in the cluster to enter a cluster transition. In this process, it would typically send a message to all cluster members, and then await messages from all cluster members acknowledging this message.

This is typical of cluster management messages. They typically have a transactional nature across the entire cluster. The most common pattern is to send a message to all nodes, and then await responses from all nodes participating in the event. This is the model IBM follows in their Phoenix HA system. A similar model is followed by TweedieCluster barrier services. If you combine this observation with the information that heartbeat information is typically broadcast to the entire cluster, it becomes readily apparent that by far the majority of high–availability packets are sent to the entire cluster. To give some sense of how rarely unicast packets occur, it would be surprising if as many as 1 packet in 100 in a normally–operating HA system is a unicast packets.

Communication Reliability

Cluster communications are the backbone around which one builds a high–availability system. If they are not reliable, then it is quite obvious that the cluster itself will will not be reliable. Indeed, there is a whole class of problems around cluster partitioning (also called split–brain syndrome) which are made significantly less likely by good cluster communications.

It is normally considered desirable to detect a cluster node failure within seconds. For example, in Microsoft's WolfPack system, the time to discover a node is dead is less than 5 seconds. In practice, this means that heartbeat messages should be delivered in significantly less than this amount of time, *even in the presence of single communication failures.* Unfortu nately, this is difficult with routing protocols, which are generally tuned to less stringent requirements, and aren't keyed to providing this information to an integrated system. Routing protocols also only deal with IP packets, which don't include "raw" serial ports. Since the code avoids relying on routing protocols for failover, potentially complex interactions between the configuration of a cluster's internal communication channels and the configuration of the surrounding network are minimized.

Moreover, in HA systems, one also wants to know that the backup links are also still working, and report this information to the cluster administration system. This significantly reduces the chances of multiple failures from which the cluster cannot recover. For example, if someone unplugs a backup communications cable and a month or two later, the primary communications link fails, the system will be unable to communicate. This failure to communicate would have been completely avoidable by reporting the failure of backup links as well as the failure of active communication links. In this way, the backup link can be repaired before the primary link also fails.

Many cluster systems also implement heartbeat mechanisms which work on independent communication methods (for example, raw serial ports) in addition to supporting ethernet for heartbeats. This is considered a best current practice, and is strongly recommended by the Linux–HA HOWTO **[Milz99]**. The reasons for this are several fold:

- Failures in the IP communication subsystem are unlikely to affect the serial subsystem
- Serial ports do not require complex external equipment or external power
- Serial ports are simple devices and very reliable in practice.
- Serial ports can be easily dedicated to cluster communications, and are not subject to significant variability in message delivery latency due to exponential backoff algorithms required by CSMA/CD media like ethernet.
- It is difficult to accidentally unplug a properly screwed–in serial connector.

However, in spite of these advantages, it should be understood that serial port communication is less well–suited for large clusters, and should be viewed as simply another tool in the tool box, to be used where it is appropriate. Large clusters may need very high speed UARTs, or nearest–neighbor heartbeat techniques which minimize required bandwidth.

Summary of Cluster Communications Observations

- Heartbeats (keepalives) are the overwhelming majority of non–application cluster control messages.

- The overwhelming majority of all cluster messages go to all cluster members
- Cluster transition messages commonly consist of a cluster–broadcast message, with a set of unicast results.
- Reliable communications are essential in a cluster
- Single communications failures should not disrupt cluster communications even momentarily
- Backup communications methods should be frequently verified for correct operation, and failures reported through the administration interface.
- Multiple, independent communication paths should be supported by the software, serial ports being the most common alternative choice
- In general, simple methods are preferred to complex methods

These are the primary considerations which led to the communications design which *heartbeat* implements.

Major Heartbeat Design Decisions

- Support many types of communication protocols, including non–IP protocols like raw serial ports
- Send all messages across all communications paths all the time. Report link failures, even when redundant links are still working.
- Send all messages to all nodes all the time. Ignore messages for other nodes.
- Support reliable multicast messaging
- Use heartbeat messages as keepalives on the communications links in the design of the reliable messaging.

Although it is apparent at this point why most of these design decisions were made, the last design decision is not yet obvious from the text presented above. To fully understand the what this means and why it was chosen, a more full exposition of the multicast protocol must be given. This will be given in the next section.

The particular type of serial port implementation chosen configures the serial ports in a bidirectional ring, similar to a FDDI ring. In this configuration, the number of nodes in the cluster are limited by the speed the serial ports can be run. For current message sizes, and links running at 56 Kbits/sec, this limits the maximum size of clusters that rely on serial ports to something over 30 nodes. This is a fairly respectable cluster size. Of course, care must be taken with system placement or the wiring of such a serial ring will become unmanageable long before this limit is reached.

Heartbeat's Reliable Multicast Protocol

Protocols have a number of characteristics which can be used to describe them. The current heartbeat protocol has the following characteristics:

- Multicast–aware
- Guaranteed packet delivery
- Packet ordering is not guaranteed
- Flow or congestion control is not provided

Although the reasons for the first two decisions are readily apparent from the nature of a high–availability cluster as discussed above, the latter two are not so immediately obvious. At this point, packet ordering is not required because of the strict request/response nature of the cluster management functions which might be put on top of it. This also obviates the need for flow control, since further packets are not typically sent until responses are received from previous packets.

There are basically two techniques described in the literature **[Weiss00]**, **[Dan94]**, **[Ram87]** for designing multicast protocols:

- Sender–initiated
- Receiver–initiated

In sender–initiated multicast protocols, the receivers of data typically send acknowledgments for packets. The sender then maintains timers and retransmits packets for which acknowledgments are not received within some fixed period of time. This class of protocols is subject to flooding senders with acknowledgments with every single packet sent. P. B. Danzig **[Dan94]** designates this flooding phenomenon as sender (or ACK) implosion. The fact that it occurs consistently with every packet sent is burdensome and makes this technique unsuitable for use in *heartbeat*, since it occurs at every machine with every packet, and could double the number of packets sent. Because of timing considerations, media collisions are very likely to be generated in large quantities as well.

In receiver–initiated protocols, the receivers of communications are responsible for detection of errors. In this scheme, sequence numbers are used to detect packet loss. When a lost packet is detected, the receiver requests that the sender retransmit the packet. In this method, senders are subject to being flooded with NACKs (NACK implosion) if the packets do not reach any receivers. This can lead to a high load on the sender, and excessive retransmissions. In 1987 Ramakrishnan et al. proposed a scheme to avoid NACK implosion **[Ram87]**. In this scheme, retransmissions are limited by timers.

A variant of this scheme is implemented in *heartbeat*. In *heartbeat*, each receiver requests a packet's retransmission no more than once per second, and in turn, each sender will re–transmit (by cluster broadcast) each packet no more than once per second as well. In this way, NACK implosion is strictly limited. HA control protocols do not have many of the requirements of some of the secure multimedia streaming protocols such as RTP **[Weis00]**.

In an HA cluster, non–heartbeat control messages are rare. In some cluster management structures, no control messages are sent until a failure (cluster transition) occurs. This means that it could be many months or even years between messages. This can make detection of lost packets by sequence numbers problematic. As a result, it is quite useful to transmit the heartbeat messages using the same communication channel as the control messages, and share the same sequence numbers. This comes from the fact that heartbeat messages are transmitted on a frequent, regular schedule. In this way strong bounds are placed on the amount of time which might elapse before a lost packet is detected. If the heartbeat messages did not ride across the same channel as normal messages, this would be more complex to guarantee.

This provides significant synergy between heartbeat communication and higher–level messages, and simplifies the implementation of the protocol. Indeed, this approach has been quite effective, and has limited the corresponding protocol code in *heartbeat* to a few hundred lines of code.

Heartbeat's Authentication Scheme

Since cluster members must be able to trust each other completely, a cluster communication system needs to have either physically secure communications, or communications which cannot readily be spoofed. Depending on the cluster manager, it is easily possible that one node in the cluster might ask another node to stop serving a particular IP address, or shut down completely, or reboot, or perform various actions with serious consequences. In some respects, allowing a node to masquerade as a cluster member is tantamount to letting it be root on all of the cluster members. A cluster communications channel then becomes a way to crack a machine, and effectively become root. *Heartbeat*'s design intent includes the idea of avoiding adding another route to crack the systems in a cluster. To address this issue *heartbeat* digitally signs every packet, and implement a few precautions in the protocol code to deter replay attacks.

It comes supplied with the following default signature algorithms:

- 32–bit CRC (for physically secure networks only)
- MD5
- HMAC–SHA1 (believed to be the most secure of the three)

As described previously, an HA cluster is a continuously–running multicast system. Unlike many multicast systems, it is largely symmetric, with every node sending approximately the same number of messages, and most of them going to the entire set of members. One of its unique features is that the communication layer is intended to run indefinitely, hopefully for many years without interruption. This includes the need to be able to do various kinds of maintenance operations without ever taking down the entire cluster. These things should include:

- Changing keys
- Changing authentication methods
- Adding new authentication methods

These properties can be problematic, and to the author's knowledge are not all solved by any standard multicast protocol. The next section elaborates on *heartbeat*'s solution to these problems, on the details of the authentication protocol, and how, together with the low level protocol, various kinds of attacks are made more difficult. It is worth noting that one of the reasons for publishing the details of this protocol is that it is hoped that feedback will be received which will help further improve the security and authentication in *heartbeat*.

The problem of Changing Keys

As noted above, it is necessary to be able to change keys, key types, and even add new authentication methods without taking the entire cluster down. In order to implement this, a key file called *authkeys* is used which contains the following information:

- signature method and key to sign out–going packets with
- signature methods and keys which will be accepted on incoming packets

Note that each packet is signed with a single signature type, but each node will authenticate incoming packets which are signed with any of set of signature types. This allows one to change keys gracefully in a system by following the steps below:

1. Initial state. Every node signs with the "old" key, and only accepts the "old" key.
2. Distribute a new authkeys file to each machine which signs with the "old" key, but accepts the old and new keys
3. Activate the new authkeys on each machine, and wait for it to be activated on each machine.
4. Distribute a new authkeys file to each machine which signs with the "new" key, but accepts the old and new keys
5. Activate the new authkeys file on each machine, and wait for things to "settle out".
6. Distribute a new authkeys file to each machine which signs with the new key and accepts only the new key.
7. Activate the new authkeys file.

There is only one detail not well explained by the above description. What is the settling in–terval, and why is it there? The settling interval occurs because packets signed with the old key may still be present in the communications sys–tem, and unless sufficient time is allotted, they may be (re)transmitted and not be accepted by the other members of the cluster. How long this interval should be depends on the details of how the reliable multicast protocol works. In the current implementation, 100 seconds is suffi–cient for this step in the worst possible case. This settling interval could be eliminated by re–signing all packets saved for possible retransmission with the new key.

Adding new authentication methods can be added to *heartbeat* without recompiling the bi–nary, because the authentication methods are now dynamically linked into the code, and will be loaded when it needs to do so without re–starting the program.

Authentication Information in Packets

Each packet is signed with a field called the "auth" field. This field contains two subvalues: a number representing the authentication method along, with the signature value (a string) The number representing the authenti–cation method does not have a fixed mapping to a particular authentication method. Attackers who see the packet cannot readily tell which authentication method is being used. If more methods are added, this will add more difficulty to the attackers job. If an individual site wishes to add security–through–obscurity, one can al–ways add some undocumented methods to make the job of brute force attack of the key space somewhat more difficult for the amateur cryp–tographer.(or script kiddee). As an additional benefit, the authentication methods also detect packets which have been corrupted by simple media errors, allowing the *heartbeat* protocol to reliably run across media like serial ports which do not verify data integrity themselves.

Replay Attacks

If an attacker can mount an active attack (sniffing and injecting packets) on the heartbeat subnet, then they can initiate a replay attack, in which properly authenticated packets which were previously sent are resent, with potentially serious effects. In a replay attack, the attacker sniffs packets from the heartbeat subnet, then resends them at an opportune time later, so that those packets are taken as genuine. This can cause the cluster to give up resources, or shut down, or take any number of actions which might have been appropriate at some point in the past, but which should not be carried out now under the control of an attacker.

Initially, after putting in the authentication code, the possibility of replay attacks was ig–nored for the following reasons:

- Very few messages have the possibility of having any impact if replayed. (they occur only rarely)
- Sniffing the local subnet was thought to be difficult in practice for a cluster subnet.

However, others made convincing arguments why this protection was too weak:

- It is possible through various attacks to cause one of the cluster member to crash us–ing one of the known denial of service attacks, creating a situation where messages with serious consequences are generated on demand.

It is desirable to add multicast heartbeats to *heartbeat*, in which case there are potentially many more places to sniff the traffic than there are in the broadcast case.

These arguments were found to be convincing. Something needed to be done. In the original code, resetting sequence numbers was simply assumed to be a link reset, so an attacker could capture packets around a link reset (which almost always involve serious actions) and replay them. As a result, a change was devised to the packet management code which is believed to make replay attacks impossible. Each time a sequence number reset is performed, the new version of *heartbeat* increments an instance number. The handling of packet sequence and instance numbers fall into the following categories:

- Packets with a higher sequence number than seen before and a current instance number: treated as received
- Packets known to be missing: treated as received
- Packets with new instance numbers: treated as a protocol restart.
- Recent, duplicate packets: ignored
- Packets with "old" sequence numbers or instance numbers: ignored and flagged as a possible replay attack

Heartbeat Message Format

It is the purpose of a high–availability system to be available without interruption for years, through hardware and software upgrades. Consequently, one of the tasks that a cluster system must undertake is on–the–fly upgrades. Towards this end, it is necessary that old versions of the software should be able to accept messages from new versions of the software, and ignore fields in the messages which they do not understand. This is an undertaking of moderate difficulty, requiring careful thought on the design of new messages to be sent in the cluster and how they will be interpreted by the old software.

In this design, a communications system like *heartbeat* can be of some help. Some message formats are designed to easily allow software to ignore fields they don't understand, and yet still find the information they need in the fields they do understand. Towards this end, messages in the heartbeat system are designed in a fashion similar to the environment strings which UNIX™[2] supplies to processes.

In this format, each message consists of a set of ASCII (name, value) pairs. New message formats most commonly add new (name, value) pairs. In some cases, it is desirable to effectively change the semantics of a given name. When this situation occurs, there are various techniques available to handle it. One of the most common ones is to have the new version of the software continue to supply the old name semantics through the old name, and redundantly supply a new name, and the new semantics associated with it through the new (name, value) pair.

Although there are various techniques to deal with this situation, this message format has the advantage of being very simple, easy to understand, and yet very flexible. As a bonus, it is a simple matter to provide the contents of a message to a shell script.

The message format *heartbeat* adopted is simple, and easy to understand, but has a few disadvantages. ASCII data is bulky, and having names in every message makes it more so. This extra bulk is primarily of concern for heartbeat messages, which constitute the majority of data in the cluster communication. This has been made a little less problematic by choosing short field names for the fields found in heartbeat messages. This trades message bulk off against a little obscurity for heartbeat messages. With strong authentication, current heartbeat messages are approximately 150 bytes in size. It is believed that this size of message is small enough to justify the design chosen.

Heartbeat API

The initial implementation of *heartbeat* had a very simple prototype API for communication with management layers of the cluster. In this API, each cluster message caused the invocation of a process to which the message was given. S imple rules were used for filtering out heartbeat messages in which no state changed (i.e., weren't associated with a node entering or leaving the cluster). This interface is extremely effective and easy to put together scripts to manage a 2–node cluster. This prototype communication mechanism proved adequate for

2 UNIX is a trademark of SCO.

some simple purposes, and allowed easy assembly of a very basic (some might say crude) cluster manager for two nodes.

However, a more sophisticated cluster management functions need more state information, and need to be continuously running in order to negotiate with other nodes. This is a critical need for clusters with greater than two nodes. The prototype API fell short in many respects. These include:

- Ability to support more than one client process
- Ability to monitor and query the current state of the cluster
- Ability to adequately isolate the client process from the implementation details
- Ability to communicate with a continuously running process.
- Ability to write an independent communications debugger
- fork/exec is unreliable under heavy load

Since the prototype API was designed as a vehicle for demonstrating *heartbeat*, none of these limitations came as any surprise. At the time of this writing, this new heartbeat API has been implemented and is now in the testing phase. This API provides the following basic services:

- Obtain node and link status
- Observe changes in node and link status
- Send reliable messages to other nodes
- Receive reliable messages from other nodes
- Keep multiple users of the API from interfering with each other.

It is expected that this API will allow *heartbeat* to become useful in a wide variety of circumstances, ranging from normal HA operations, through integrating with other cluster managers, through integrating with cluster file systems.

STONITH Implementation

On the linux–ha mailing list, the term STONITH has been used to describe a technique for I/O fencing which allows one to guarantee exclusive use of a set of shared resources. STONITH stands for **S**hoot **T**he **O**ther **N**ode **I**n **T**he **H**ead. This technique allows cluster systems to safely use shared disk arrangements. An API has been defined for this technique, and an implementation was created. Although it is useful now, it will be more useful in the future, with a new cluster management infrastructure. At this point, the API and implementation will appear in *heartbeat* and several other open source cluster managers.

Future Directions

There are many possibilities on the horizon for *heartbeat*. These possibilities have been considerably broadened by the recently introduced API. These include.

True multicast

Currently, *heartbeat* does all of its UDP communication using broadcast packets. This limits the cluster to being all on one subnet, and causes unnecessary interrupts on machines which are on the subnet but not part of the cluster. It would be good to implement multicast communications in order to alleviate these concerns.

Automated key distribution and management

At the present time, it is necessary for all the members of the cluster to have their authentication data updated manually. It would be desirable to have a way to distribute keys across the cluster, perhaps using openssh or some similar mechanism.

Integration with LinuxFailSafe

SGI and SuSE have made SGI's FailSafe product available on Linux as an open source software package **[Vas00]**. Although Linux-FailSafe currently has heartbeat and membership mechanisms of its own, there are enough things that *heartbeat* does better, that it is expected to prove desirable to replace portions of FailSafe with *heartbeat*.

GUI tools

There are at least two efforts underway to add a GUI configuration and status front end to *heartbeat*. Conectiva has written a linuxconf module for *heartbeat*, and David Martinez has prototyped a GUI configuration tool for it **[Mar00]**.

Cluster Consensus Membership

Heartbeat provides only a local view of cluster membership. That is, it only knows about or is aware of each node's own individual view of the cluster membership. This is adequate for some purposes, but inadequate for many. If the local medium has no communication asymmetries, and the same configuration, then there is no difference between the local and the global view.

However, if communication is impeded due to hub or switch bugs or routed multicasts, then it is possible for different members of the cluster to have differing views of what the membership of the cluster actually is. It is important that every member of the cluster have the same view of cluster membership as every other member. It is necessary to have a consensus membership layer which then shares cluster membership information across the cluster, so that each machine has the same view of the cluster membership.

Barrier API implementation

Certain kinds of clusters use the model of barriers in their implementation. The Tweedie-Cluster is an example of such a cluster **[Twe00]**. According to Tweedie, a barrier provides a guaranteed synchronization point in distributed processing which is valid over all nodes in the cluster: it strictly divides time into a pre–barrier and a post–barrier phase. The barrier may not necessarily complete at exactly the same time on every node, but there is absolute guarantee that the barrier will not complete on any node unless all other nodes have begun the barrier. The barrier API he describes could easily be implemented on top of the heartbeat API.

Phoenix n–phase transactions

IBM's Phoenix Cluster technology has the concept of providing a tool kit to implement generalized n–phase transactions to achieve synchronization across the cluster **[Pfi98]**. These are similar in function to the barrier API described above, but are more general. In this model, a transaction is begun in which a machine names a next state. Each node participating in the transaction then sends a message nominating the next state. When all nodes have reported the next state, then the transaction proceeds to the next phase. If the membership changes or any node nominates a differing next state during this time, the transaction is aborted. If this does not happen, the transition to the next state is accomplished, and the process is repeated until a final state is achieved. The idea of a generalized synchronization mechanism seems like a very good idea, and one worth emulating in the Linux–HA environment. This mechanism could be implemented on top of a barrier API, or using the heartbeat API directly.

Dynamic loading of modules

Much of the code in *heartbeat* consists of modules which are invoked through table lookups. These modules provide services which it would be desirable to add dynamic loading so that new modules could be added to *heartbeat* without restarting the services. In order to ensure reliable heartbeat services in the presence of ill–behaved applications, *heartbeat* runs locked into memory, and at high priority. Because of this, it is desirable to minimize the amount of memory which *heartbeat* used. Replacing the static linking which *heartbeat* uses with dynamic linking would help in this effort.

It is anticipated that *heartbeat* would benefit from dynamically loading at least the authentication methods, heartbeat media drivers, and STONITH implementations. At this point in time, the implementation of dynamic loading is being tested.

Ping membership

For two–node systems, it is often desirable to have a resource which can be used as a quorum resource, so that one can guarantee that only one of two partitions of a cluster can be active at a time. Since Linux–HA systems are intended to achieve very low deployment costs, it is undesirable to add a third node solely for the purpose of breaking ties. As a result, it would be desirable to allow relatively unintelligent devices which are already present on the customer's site to act as pseudo–cluster members. For these devices to participate as members, a new class of membership would be added to *heartbeat*: ping membership. Whenever the local system would send out its heartbeat, a ping packet would be sent to the pseudo–member. Whenever a ping response is received, it would be translated into a heartbeat message. The result of this is that such devices could effectively act as tie–breakers in the case of 2–node quorum systems. Candidates for such devices include switches, routers, and ethernet–accessible intelligent power controllers or other pingable devices. There are problems potentially associated with this approach: It is possible (through ARP cache problems for example) for each of two nodes to be able to communicate with a common endpoint, yet be unable to communicate with each other. At this time, an implementation of this technique has been prototyped and further research will be done to see if there is a variant on this technique which would be helpful in solving these difficulties.

External Cluster Manager

Heartbeat's current cluster manager was originally prototyped to demonstrate *heartbeat*.

It is unsophisticated and very limited in its function. However, the introduction of the heartbeat API opens up the possibility of pro–viding a "real" cluster manager, or a series of them. It is expected that *heartbeat* will drop its primitive cluster management facility in favor of external cluster management. As a result, this current implementation is not discussed here.

Design Retrospective

The previous sections describe the positive reasons why the design decisions were made without significant references being made to the corresponding costs or disadvantages. This section will explore these costs, and comment on the design of these key areas. Some of the areas addressed here were pointed out by the readers of the linux–ha–dev mailing list.

Communication using PPP

Initially, the *heartbeat* code was implemented directly on top of the serial device. Later on it was suggested by Stephen Tweedie and Alan Cox that PPP might be a better choice for using the serial ports. This seemed worth trying, so it was implemented. Unfortunately, there were several problems which resulted. These prob–lems include: PPP unreliability, slow startups, code complexity, and PPP hangs. In a small percentage of cases, PPP would not start at all, and had to be killed and restarted, sometimes on both ends. Even when it does start correctly, PPP can take up to seven seconds to start. This makes hangs slow to detect, and delays starting up the cluster. Sometimes, for no apparent rea–son, PPP would stop transmitting in one direction for no apparent reason and with no obvious symptoms. These bugs and behaviors in PPP make it a poor choice for use in highly reliable systems. These problems exist today, and have existed for more than a year in the Linux PPP implementation. The workarounds for all these problems make the PPP transport module in *heartbeat* about twice as large and complex as the corresponding code for any other medium, in addition to being unreliable.

Code Size

Although the *heartbeat* code has proven quite robust, various people have asked questions about its size. On i386 Linux, the object size is approximately 128K including all *heartbeat* media, authentication methods, STONITH methods, the API, and the code for the proto–type cluster management function. This breaks down to lines of commented source like this: 4000 lines for core function, 800 lines for mes–saging functions, 600 for authentication, 1000 for parsing configuration files, 2000 lines for handling different transport media (over half of which is PPP code), 1400 for the heartbeat API. At this point, it seems to be eminently main–tainable. The recently–added ability to dynamically load modules will minimize mem–ory usage while allowing for more growth in capabilities.

Flexibility

It has proven to be quite simple to add new *heartbeat* media types, authentication types, message types and features to *heartbeat*. Much of the code is table–driven, and is overall quite straightforward and simple to extend and change. Indeed, one of the unexpected prob–lems associated with writing this paper has been that many things have changed and been imple–mented while it was being written, causing it to have to be updated continually.

Bandwidth usage

One of the concerns which been expressed concerning the design of *heartbeat* is that its combination of ASCII message format makes for verbose heartbeat messages. The concern is specifically that it might become a bandwidth hog as cluster sizes grow.

The formula for computing bandwidth used by *heartbeat* for N nodes on an unswitched net–work is as follows:

$$B_{hb}= S_{hb} * 8_{\text{bits/byte}}\ R_{hb} * N$$

Where B_{hb} is the bandwidth consumed in bits/sec, S_{hb} *is the size of a heartbeat's packet in bytes, R_{hb} is the rate of heartbeat per cluster node, and N is the number of nodes in a cluster. A common heartbeat rate is 1 packet/sec, heart–beat packets average around 150 bytes. If a cluster has 1000 nodes with these other charac–teristics, then the B_{hb} for such a system is 1.2 x 10^6 bits/sec. This is approximately 1.2% of the bandwidth available on an unswitched 100 Mbit network. It is clear from this calculation that fears about *heartbeat* bandwidth consumption are unfounded for realistic–sized clusters. On the other hand, it is also clear from this calcula–tion that this same bandwidth would significantly overwhelm a normal PC serial connection.

Reliability

Although released versions of *heartbeat* have had a few bugs, they have largely exhibited themselves consistently on system startup or not all. The result of this is that *heartbeat* has proven itself to be quite reliable in actual field usage.

Protocol Limitations

The current heartbeat protocol guarantees that packets are delivered to every active node, or that the client is notified of the death of the nodes to which it isn't delivered subject to resource limitations. However, these packets are not guaranteed to be delivered in any particular order, nor is any flow control provided. The packet delivery order problem is relatively easily solvable should that prove desirable. The flow control issue is not expected to be a problem in the domain for which this protocol is designed (high–availability control messages). However, if a client were sufficiently ill–behaved it is possible that it could exhaust packet retransmission facilities, resulting in a packet not being delivered. In practical tests with well–behaved clients and extraordinary packet loss rates (90% on transmission, and 90% on reception), this behavior could not be induced in many hours of testing.

Security

Because it was designed from the ground up to operate in the cracker–rich internet environment as an open source Linux package, it is one of the more security–conscious high–availability implementations available. Although it makes no attempt to encrypt data because of repressive US laws on encryption, it does make good use of strong authentication protocols.

Perhaps too successful

One of the most interesting and curious criticisms of *heartbeat* is that it has been too successful. What was meant by this was that *heartbeat* solved a wide–enough class of problems sufficiently well that there was little motivation to create an alternative solution, even though *heartbeat* itself was quite limited. In particular, the original heartbeat API was incapable of supporting a sophisticated cluster management infrastructure, so that until the new API was added, it was impossible to go beyond two nodes in the cluster. Now that this API is available, it is possible to write a better cluster management services. Now, *heartbeat*'s past success can now serve as a springboard for future enhancements and greater usefulness rather than serving as a source of criticism.

Is *heartbeat* misnamed?

Because of the fact that it treats providing heartbeat services as a subcase of cluster communications, some have said that *heartbeat* is misnamed. It is not simply a heartbeat mechanism, but is best thought of as a robust low–level cluster communications mechanism which provides notification when nodes join and leave the communication channel. In this sense, the heartbeat could perhaps even be thought of as a side–effect of the reliable communications protocol.

Conclusion

Heartbeat's technique of providing heartbeats as an integral part of a cluster communications channel has proven straightforward and to implement and maintain. its use of ASCII message formats has proven quite flexible, and added to the ease of debugging. The addition of an API is pushing it into a new stage of growth and usefulness in roles where it provides only cluster communication and membership services. This is expected to make it more useful than ever, and allow many different applications and cluster management functions to interface with it.

Acknowledgments

Thanks to the members of the Linux–HA and linux–ha–dev mailing lists who have used and critiqued and questioned aspects of *heartbeat*. In this regard, the author offers special thanks to Jerome Etienne. In many ways, Jerome inspired this paper by persistently asking questions about *heartbeat*, its design decisions and questioning their wisdom. Without Jerome's persistence and energy, this paper would likely not have been written. Additionally, Jerome commented on extensively on several drafts of this document. The following people also provided useful comments on drafts of this document which improved it considerably: Peter Badovinatz, Simon ("horms") Horman, Neal McBurnett, John Schmaus and Stephen Tweedie. The author would also like to thank Neal McBurnett of AVAYA (formerly Bell Labs) for his help in designing the authentication scheme used by *heartbeat*. Additionally, many, many people have contributed to *heartbeat*, by writing code for it, or testing it, or providing documentation for it. Without them, *heartbeat* itself would not be nearly the program it is now.

To Learn More

The Linux–HA web site can be found at **[Rob00]**. *Heartbeat* can be downloaded (in source or RPM format) from the Linux–HA web site download page at: http://linux–ha.org/download/. Information on subscribing to the various Linux–HA mailing lists can be found on the contact page at: http://linux–ha.org/contact/. The Linux FailSafe project is described in detail in **[Vas00]**.

References

[Dan94] Danzig, P. B.: "Flow Control for Limited Buffer Multicast", IEEE Transactions on Software Engineering, Vol 20, No. 1, January 1994, pp. 1–12

[Milz99] Milz, Harald: "The Linux High Availability HOWTO". http://metalab.unc.edu/pub/linux/ALPHA/linux–ha/High–Availability–HOWTO.html

[Mar00] Martinez, D.: http://linux–ha.org/screenshots.html

[Phi98] *In Search of Clusters*, by Gregory F. Pfister, 2nd Edition 1998, Prentice Hall PTR

.[Ram87] Ramakrishnan, S., Jain, B. N.: "A Negative Acknowledgment with Periodic Polling Protocol for Multicast over LANs". In: Proc. IEEE INFOCOM '87, March 1987, S. 502–511.

[Rob00] Robertson, A. L.,: "The High–Availability Linux Project". Http://linux–ha.org/

[Twe00] Tweedie, S. C.,: "Barrier Operations". http://linux–ha.org/PhaseII/WhitePapers/sct/barrier.txt

[Vas00] Vasa, M.,: "The Linux Fail Safe Project". http://oss.sgi.com/projects/failsafe/

[Wei00] Weis, R., Geyer, W, Kuhmünch, C., "Architectures for Secure Multicast Communication", In: Proc. SANE 2000 System Administration and Networking Conference, May 22–25, 2000., pp. 63–91.

PVFS: A Parallel File System for Linux Clusters*

Philip H. Carns   Walter B. Ligon III
Parallel Architecture Research Laboratory
Clemson University, Clemson, SC 29634, USA
`{pcarns, walt} @parl.clemson.edu`

Robert B. Ross   Rajeev Thakur
Mathematics and Computer Science Division
Argonne National Laboratory, Argonne, IL 60439, USA
`{rross, thakur} @mcs.anl.gov`

Abstract

As Linux clusters have matured as platforms for low-cost, high-performance parallel computing, software packages to provide many key services have emerged, especially in areas such as message passing and networking. One area devoid of support, however, has been parallel file systems, which are critical for high-performance I/O on such clusters. We have developed a parallel file system for Linux clusters, called the Parallel Virtual File System (PVFS). PVFS is intended both as a high-performance parallel file system that anyone can download and use and as a tool for pursuing further research in parallel I/O and parallel file systems for Linux clusters.

In this paper, we describe the design and implementation of PVFS and present performance results on the Chiba City cluster at Argonne. We provide performance results for a workload of concurrent reads and writes for various numbers of compute nodes, I/O nodes, and I/O request sizes. We also present performance results for MPI-IO on PVFS, both for a concurrent read/write workload and for the BTIO benchmark. We compare the I/O performance when using a Myrinet network versus a fast-ethernet network for I/O-related communication in PVFS. We obtained read and write bandwidths as high as 700 Mbytes/sec with Myrinet and 225 Mbytes/sec with fast ethernet.

*This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38, and in part by the National Aeronautics and Space Administration, under Research Grant NAG-5-3835.

1 Introduction

Cluster computing has recently emerged as a mainstream method for parallel computing in many application domains, with Linux leading the pack as the most popular operating system for clusters. As researchers continue to push the limits of the capabilities of clusters, new hardware and software have been developed to meet cluster computing's needs. In particular, hardware and software for message passing have matured a great deal since the early days of Linux cluster computing; indeed, in many cases, cluster networks rival the networks of commercial parallel machines. These advances have broadened the range of problems that can be effectively solved on clusters.

One area in which commercial parallel machines have always maintained great advantage, however, is that of parallel file systems. A production-quality high-performance parallel file system has not been available for Linux clusters, and without such a file system, Linux clusters cannot be used for large I/O-intensive parallel applications. We have developed a parallel file system for Linux clusters, called the Parallel Virtual File System (PVFS) [33], that can potentially fill this void. PVFS is being used at a number of sites, such as Argonne National Laboratory, NASA Goddard Space Flight Center, and Oak Ridge National Laboratory. Other researchers are also using PVFS in their studies [28].

We had two main objectives in developing PVFS. First, we needed a basic software platform for pursuing further research in parallel I/O and parallel file systems in the context of Linux clusters. For this purpose, we needed a stable, full-featured parallel file system to begin with. Our second objective was to meet the need for a parallel file system for Linux clusters. Toward that end, we designed PVFS with the following goals in mind:

- It must provide high bandwidth for concurrent read/write operations from multiple processes or threads to a common file.
- It must support multiple APIs: a native PVFS API, the UNIX/POSIX I/O API [15], as well as other APIs such as MPI-IO [13, 18].
- Common UNIX shell commands, such as `ls`, `cp`, and `rm`, must work with PVFS files.
- Applications developed with the UNIX I/O API must be able to access PVFS files without recompiling.
- It must be robust and scalable.
- It must be easy for others to install and use.

In addition, we were (and are) firmly committed to distributing the software as open source.

In this paper we describe how we designed and implemented PVFS to meet the above goals. We also present performance results with PVFS on the Chiba City cluster [7] at Argonne National Laboratory. We first present the performance for a workload comprising concurrent reads and writes using native PVFS calls. We then present results for the same workload, but by using MPI-IO [13, 18] functions instead of native PVFS functions. We also consider a more difficult access pattern, namely, the BTIO benchmark [21]. We compare the performance when using a Myrinet network versus a fast-ethernet network for all I/O-related communication.

The rest of this paper is organized as follows. In the next section we discuss related work in the area of parallel file systems. In Section 3 we describe the design and implementation of PVFS. Performance results are presented and discussed in Section 4. In Section 5 we outline our plans for future work.

2 Related Work

Related work in parallel and distributed file systems can be divided roughly into three groups: commercial parallel file systems, distributed file systems, and research parallel file systems.

The first group comprises commercial parallel file systems such as PFS for the Intel Paragon [11], PIOFS and GPFS for the IBM SP [10], HFS for the HP Exemplar [2], and XFS for the SGI Origin2000 [35]. These file systems provide high performance and functionality desired for I/O-intensive applications but are available only on the specific platforms on which the vendor has implemented them. (SGI, however, has recently released XFS for Linux. SGI is also developing a version of XFS for clusters, called CXFS, but, to our knowledge, CXFS is not yet available for Linux clusters.)

The second group comprises distributed file systems such as NFS [27], AFS/Coda [3, 8], InterMezzo [4, 16], xFS [1], and GFS [23]. These file systems are designed to provide distributed access to files from multiple client machines, and their consistency semantics and caching behavior are designed accordingly for such access. The types of workloads resulting from large parallel scientific applications usually do not mesh well with file systems designed for distributed access; particularly, distributed file systems are not designed for high-bandwidth concurrent writes that parallel applications typically require.

A number of research projects exist in the areas of parallel I/O and parallel file systems, such as PIOUS [19], PPFS [14, 26], and Galley [22]. PIOUS focuses on viewing I/O from the viewpoint of transactions [19], PPFS research focuses on adaptive caching and prefetching [14, 26], and Galley looks at disk-access optimization and alternative file organizations [22]. These file systems may be freely available but are mostly research prototypes, not intended for everyday use by others.

3 PVFS Design and Implementation

As a parallel file system, the primary goal of PVFS is to provide high-speed access to file data for parallel applications. In addition, PVFS provides a clusterwide consistent name space, enables user-controlled striping of data across disks on different I/O nodes, and allows existing binaries to operate on PVFS files without the need for recompiling.

Like many other file systems, PVFS is designed as a client-server system with multiple servers, called I/O daemons. I/O daemons typically run on separate nodes in the cluster, called I/O nodes, which have disks attached to them. Each PVFS file is striped across the disks on the I/O nodes. Application processes interact with PVFS via a client library. PVFS also has a manager daemon that handles only metadata operations such as permission checking for file creation, open, close, and remove operations. The manager does not participate in read/write operations; the client library and the I/O daemons handle all file I/O without the intervention of the manager. The clients, I/O daemons, and the manager need not be run on different machines. Running them on different machines may result in higher performance, however.

PVFS is primarily a user-level implementation; no kernel modifications or modules are necessary to install or operate the file system. We have, however, created a Linux kernel module to make simple file manipulation more convenient. This issue is touched upon in Section 3.5. PVFS currently uses TCP for all internal communication. As a result it is not dependent on any particular message-passing library.

3.1 PVFS Manager and Metadata

A single manager daemon is responsible for the storage of and access to all the metadata in the PVFS file system. Metadata, in the context of a file system, refers to information describing the characteristics of a file, such as permissions, the owner and group, and, more important, the physical distribution of the file data. In the case of a parallel file system, the distribution information must include both file locations on disk and disk locations in the cluster. Unlike a traditional file system, where metadata and file data are all stored on the raw blocks of a single device, parallel file systems must distribute this data among many physical devices. In PVFS, for simplicity, we chose to store both file data and metadata in files on existing local file systems rather than directly on raw devices.

PVFS files are striped across a set of I/O nodes in order to facilitate parallel access. The specifics of a given file distribution are described with three metadata parameters: base I/O node number, number of I/O nodes, and stripe size. These parameters, together with an ordering of the I/O nodes for the file system, allow the file distribution to be completely specified.

An example of some of the metadata fields for a file `/pvfs/foo` is given in Table 1. The *pcount* field specifies that the data is spread across three I/O nodes, *base* specifies that the first (or base) I/O node is node 2, and *ssize* specifies that the stripe size—the unit by which the file is divided among the I/O nodes—is 64 Kbytes. The user can set these parameters when the file is created, or PVFS will use a default set of values.

Table 1: Metadata example: File `/pvfs/foo`.

inode	1092157504
⋮	⋮
base	2
pcount	3
ssize	65536

Application processes communicate directly with the PVFS manager (via TCP) when performing operations such as opening, creating, closing, and removing files. When an application opens a file, the manager returns to the application the locations of the I/O nodes on which file data is stored. This information allows applications to communicate directly with I/O nodes when file data is accessed. In other words, the manager is not contacted during read/write operations.

One issue that we have wrestled with throughout the development of PVFS is how to present a directory hierarchy of PVFS files to application processes. At first we did not implement directory-access functions and instead simply used NFS [27] to export the metadata directory to nodes on which applications would run. This provided a global name space across all nodes, and applications could change directories and access files within this name space. The method had some drawbacks, however. First, it forced system administrators to mount the NFS file system across all nodes in the cluster, which was a problem in large clusters because of limitations with NFS scaling. Second, the default caching of NFS caused problems with certain metadata operations.

These drawbacks forced us to reexamine our implementation strategy and eliminate the dependence on NFS for metadata storage. We have done so in the latest version of PVFS, and, as a result, NFS is no longer a requirement. We removed the dependence on NFS by trapping system calls related to directory access. A mapping routine determines whether a PVFS directory is being accessed, and, if so, the operations are redirected to the PVFS manager. This trapping mechanism, which is used extensively in the PVFS client library, is described in Section 3.4.

3.2 I/O Daemons and Data Storage

At the time the file system is installed, the user specifies which nodes in the cluster will serve as I/O nodes. The I/O nodes need not be distinct from the compute nodes.

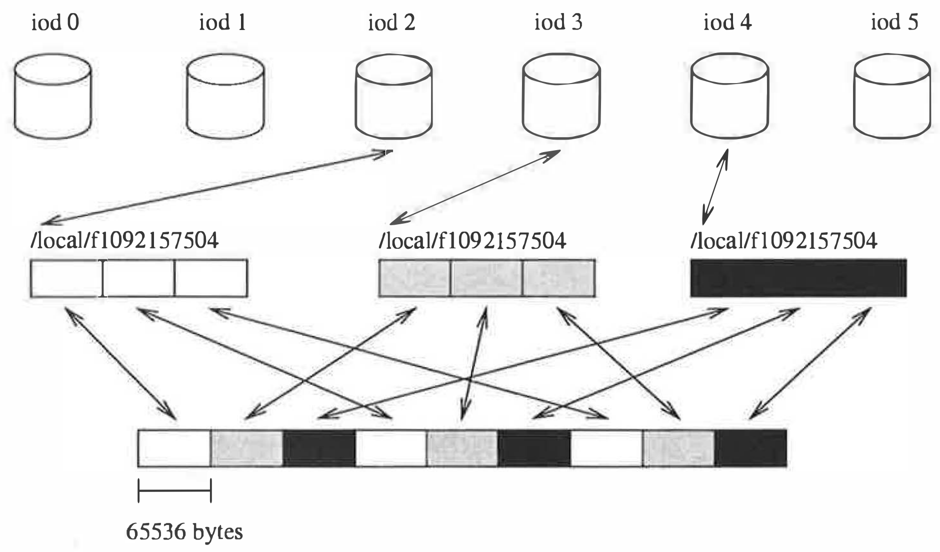


Figure 1: File-striping example

An ordered set of PVFS I/O daemons runs on the I/O nodes. The I/O daemons are responsible for using the local disk on the I/O node for storing file data for PVFS files.

Figure 1 shows how the example file `/pvfs/foo` is distributed in PVFS based on the metadata in Table 1. Note that although there are six I/O nodes in this example, the file is striped across only three I/O nodes, starting from node 2, because the metadata file specifies such a striping. Each I/O daemon stores its portion of the PVFS file in a file on the local file system on the I/O node. The name of this file is based on the inode number that the manager assigned to the PVFS file (in our example, 1092157504).

As mentioned above, when application processes (clients) open a PVFS file, the PVFS manager informs them of the locations of the I/O daemons. The clients then establish connections with the I/O daemons directly. When a client wishes to access file data, the client library sends a descriptor of the file region being accessed to the I/O daemons holding data in the region. The daemons determine what portions of the requested region they have locally and perform the necessary I/O and data transfers.

Figure 2 shows an example of how one of these regions, in this case a regularly strided logical partition, might be mapped to the data available on a single I/O node. (Logical partitions are discussed further in Section 3.3.) The intersection of the two regions defines what we call an I/O stream. This stream of data is then transferred in logical file order across the network connection. By retaining the ordering implicit in the request and allowing the underlying stream protocol to handle packetization, no additional overhead is incurred with control messages at the application layer.

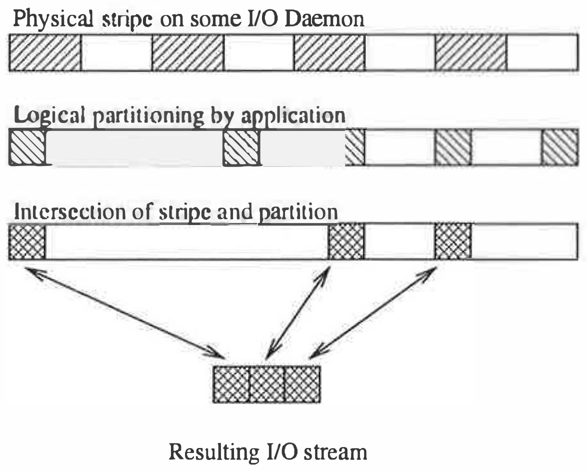

Physical stripe on some I/O Daemon
Logical partitioning by application
Intersection of stripe and partition
Resulting I/O stream

Figure 2: I/O stream example

3.3 Application Programming Interfaces

PVFS can be used with multiple application programming interfaces (APIs): a native API, the UNIX/POSIX API [15], and MPI-IO [13, 18]. In all these APIs, the communication with I/O daemons and the manager is handled transparently within the API implementation.

The native API for PVFS has functions analogous to the UNIX/POSIX functions for contiguous reads and writes. The native API also includes a "partitioned-file interface" that supports simple strided accesses in the file. Partitioning allows for noncontiguous file regions to be accessed with a single function call. This concept is similar to logical file partitioning in Vesta [9] and file views in MPI-IO [13, 18]. The user can specify a file partition in PVFS by using a special `ioctl` call. Three parameters, *offset*, *gsize*, and *stride*, specify the partition, as shown in Figure 3. The *offset* parameter defines how far into the file the partition begins relative to the first byte of the file, the *gsize* parameter defines the size of the simple strided regions of data to be accessed, and the *stride* parameter defines the distance between the start of two consecutive regions.

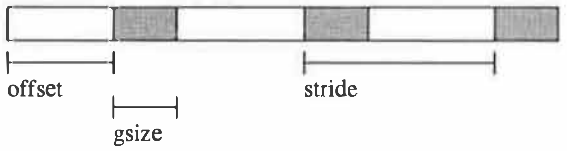


Figure 3: Partitioning parameters

We have also implemented the MPI-IO interface [13, 18] on top of PVFS by using the ROMIO implementation of MPI-IO [24]. ROMIO is designed to be ported easily to new file systems by implementing only a small set of functions on the new file system [30, 32]. This feature enabled us to have all of MPI-IO implemented on top of PVFS in a short time. We used only the contiguous read/write functions of PVFS in this MPI-IO implementation because the partitioned-file interface of PVFS supports only a subset of the noncontiguous access patterns that are possible in MPI-IO. Noncontiguous MPI-IO accesses are implemented on top of contiguous read/write functions by using a ROMIO optimization called data sieving [31]. In this optimization, ROMIO makes large contiguous I/O requests and extracts the necessary data. We are currently investigating how the PVFS partitioning interface can be made more general to support MPI-IO's noncontiguous accesses.

PVFS also supports the regular UNIX I/O functions, such as `read()` and `write()`, and common UNIX shell commands, such as `ls`, `cp`, and `rm`. (We note that `fcntl` file locks are not yet implemented.) Furthermore, existing binaries that use the UNIX API can access PVFS files without recompiling. The following section describes how we implemented these features.

3.4 Trapping UNIX I/O Calls

System calls are low-level methods that applications can use for interacting with the kernel (for example, for disk and network I/O). These calls are typically made by calling wrapper functions implemented in the standard C library, which handle the details of passing parameters to the kernel. A straightforward way to trap system calls is to provide a separate library to which users relink their code. This approach is used, for example, in the Condor system [17] to help provide checkpointing in applications. This method, however, requires relinking of each application that needs to use the new library.

When compiling applications, a common practice is to use dynamic linking in order to reduce the size of the executable and to use shared libraries of common functions. A side effect of this type of linking is that the executables can take advantage of new libraries supporting the same functions without recompilation or relinking. We use this method of linking the PVFS client library to trap I/O system calls before they are passed to the kernel. We provide a library of system-call wrappers that is loaded before the standard C library by using the Linux environment variable `LD_PRELOAD`. As a result, existing binaries can access PVFS files without recompiling.

Figure 4a shows the organization of the system-call mechanism before our library is loaded. Applications call functions in the C library (`libc`), which in turn call the system calls through wrapper functions implemented in `libc`. These calls pass the appropriate values through to the kernel, which then performs the desired operations. Figure 4b shows the organization of the system-call mechanism again, this time with the PVFS client library in place. In this case the `libc` system-call wrappers are replaced by PVFS wrappers that determine the type of file on which the operation is to be performed. If the file is a PVFS file, the PVFS I/O library is used to handle the function. Otherwise the parameters are passed on to the actual kernel call.

This method of trapping UNIX I/O calls has limitations, however. First, a call to `exec()` will destroy the state that we save in user space, and the new process will therefore not be able to use file descriptors that referred to open PVFS files before the `exec()` was called. Second, porting this feature to new architectures and operating systems is nontrivial. The appropriate system library calls must be identified and included in our library. This process must also be repeated when the APIs of system libraries change. For example, the GNU C library (`glibc`) API is constantly changing, and, as a result,

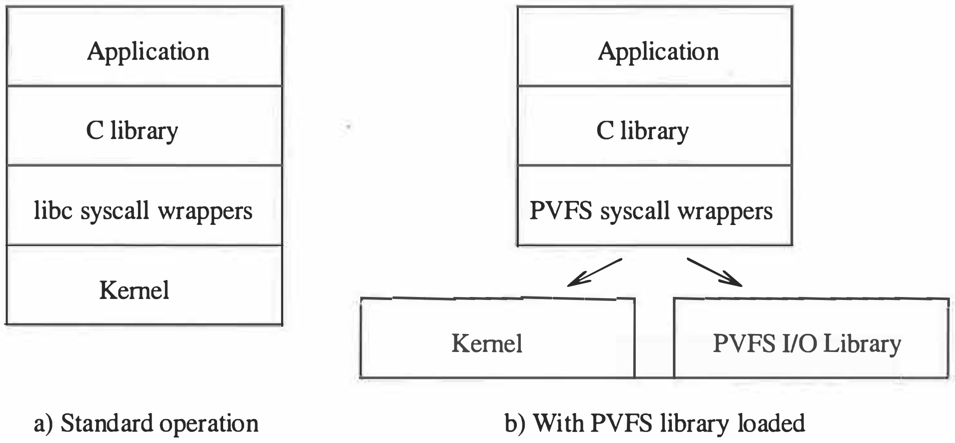


Figure 4: Trapping system calls

we have had to constantly change our code!

3.5 Linux Kernel VFS Module

While the trapping technique described above does provide the necessary functionality for using existing applications on PVFS files, the shortcomings of this method and the effort required to keep up with changes in the C library encouraged us to seek an alternative solution. The Linux kernel provides the necessary hooks for adding new file-system support via loadable modules without recompiling the kernel. Accordingly, we have implemented a module that allows PVFS file systems to be mounted in a manner similar to NFS [27]. Once mounted, the PVFS file system can be traversed and accessed with existing binaries just as any other file system. We note that, for the performance experiments reported in this paper, we used the PVFS library and not the kernel module.

4 Performance Results

We present performance results using PVFS on the Chiba City [7] cluster at Argonne National Laboratory. The cluster was configured as follows at the time of our experiments. There were 256 nodes, each with two 500-MHz Pentium III processors, 512 Mbytes of RAM, a 9 Gbyte Quantum Atlas IV SCSI disk, a 100 Mbits/sec Intel EtherExpress Pro fast-ethernet network card operating in full-duplex mode, and a 64-bit Myrinet card (Revision 3). The nodes were running Linux 2.2.15pre4. There were two MPI implementations: MPICH 1.2.0 for fast ethernet and MPICH-GM 1.1.2 for Myrinet. The kernel was compiled for a single processor; therefore, one processor on each machine was unused during our experiments. Out of the 256 nodes, only 60 nodes were available at a time for our experiments. We used some of those 60 nodes as compute nodes and some as I/O nodes for PVFS.

The Quantum Atlas IV 9 Gbyte disk has an advertised sustained transfer rate of 13.5–21.5 Mbytes/sec. The performance of the disk measured using the `bonnie` file-system benchmark [5] showed a write bandwidth of 22 Mbytes/sec and a read bandwidth of 15 Mbytes/sec when accessing a 512 Mbyte file in a sequential manner. The write performance measured by `bonnie` is slightly higher than the advertised sustained rates, perhaps because the test accessed the file sequentially, thereby allowing file-system caching, read ahead, and write behind to better organize disk accesses.

Since PVFS currently uses TCP for all communication, we measured the performance of TCP on the two networks on the cluster. For this purpose, we used the `ttcp` test, version 1.1 [29]. We tried three buffer sizes, 8 Kbytes, 64 Kbytes, and 256 Kbytes, and for all three, `ttcp` reported a bandwidth of around 10.2 Mbytes/sec on fast ethernet and 37.7 Mbytes/sec on Myrinet.

To measure PVFS performance, we performed experiments that can be grouped into three categories: concurrent reads and writes with native PVFS calls, concurrent reads and writes with MPI-IO, and the BTIO benchmark. We varied the number of I/O nodes, compute nodes, and I/O size and measured performance with both fast ethernet and Myrinet. We used the default file-stripe size of 16 Kbytes in all experiments.

4.1 Concurrent Read/Write Performance

Our first test program is a parallel MPI program in which all processes perform the following operations using the native PVFS interface: open a new PVFS file that is common to all processes, concurrently write data blocks to disjoint regions of the file, close the file, reopen it, simultaneously read the same data blocks back from the file, and then close the file. Application tasks synchronize before and after each I/O operation. We recorded the time for the read/write operations on each node and, for calculating the bandwidth, used the maximum of the time taken on all processes. In all tests, each compute node wrote and read a single contiguous region of size $2N$ Mbytes, N being the number of I/O nodes in use. For example, for the case where 26 application processes accessed 8 I/O nodes, each application task wrote 16 Mbytes, resulting in a total file size of 416 Mbytes. Each test was repeated five times, and the lowest and highest values were discarded. The average of the remaining three tests is the value reported.

Figure 5 shows the read and write performance with fast ethernet. For reads, the bandwidth increased at a rate of approximately 11 Mbytes/sec per compute node, up to 46 Mbytes/sec with 4 I/O nodes, 90 Mbytes/sec with 8 I/O nodes, and 177 Mbytes/sec with 16 I/O nodes. For these three cases, the performance remained at this level until approximately 25 compute nodes were used, after which performance began to tail off and became more erratic. With 24 I/O nodes, the performance increased up to 222 Mbytes/sec (with 24 compute nodes) and then began to drop. With 32 I/O nodes, the performance increased less quickly, attained approximately the same peak read performance as with 24 I/O nodes, and dropped off in a similar manner. This indicates that we reached the limit of our scalability with fast ethernet.

The performance was similar for writes with fast ethernet. The bandwidth increased at a rate of approximately 10 Mbytes/sec per compute node for the 4, 8, and 16 I/O-node cases, reaching peaks of 42 Mbytes/sec, 83 Mbytes/sec, and 166 Mbytes/sec, respectively, again utilizing almost 100% of the available TCP bandwidth. These cases also began to tail off at approximately 24 compute nodes. Similarly, with 24 I/O nodes, the performance increased to a peak of 226 Mbytes/sec before leveling out, and with 32 I/O nodes, we obtained no better performance. The slower rate of increase in bandwidth indicates that we exceeded the maximum number of sockets across which it is efficient to service requests on the client side.

We observed significant performance improvements by running the same PVFS code (using TCP) on Myrinet instead of fast ethernet. Figure 6 shows the results. The read bandwidth increased at 31 Mbytes/sec per compute process and leveled out at approximately 138 Mbytes/sec with 4 I/O nodes, 255 Mbytes/sec with 8 I/O nodes, 450 Mbytes/sec with 16 I/O nodes, and 650 Mbytes/sec with 24 I/O nodes. With 32 I/O nodes, the bandwidth reached 687 Mbytes/sec for 28 compute nodes, our maximum tested size. For writing, the bandwidth increased at a rate of approximately 42 Mbytes/sec, higher than the rate we measured with `ttcp`. While we do not know the exact cause of this, it is likely that some small implementation difference resulted in PVFS utilizing a slightly higher fraction of the true Myrinet bandwidth than `ttcp`. The performance levelled at 93 Mbytes/sec with 4 I/O nodes, 180 Mbytes/sec with 8 I/O nodes, 325 Mbytes/sec with 16 I/O nodes, 460 Mbytes/sec with 24 I/O nodes, and 670 Mbytes/sec with 32 I/O nodes.

In contrast to the fast-ethernet results, the performance with Myrinet maintained consistency as the number of compute nodes was increased beyond the number of I/O nodes, and, in the case of 4 I/O nodes, as many as 45 compute nodes (the largest number tested) could be efficiently serviced.

4.2 MPI-IO Performance

We modified the same test program to use MPI-IO calls rather than native PVFS calls. The number of I/O nodes was fixed at 32, and the number of compute nodes was varied. Figure 7 shows the performance of the MPI-IO and native PVFS versions of the program. The performance of the two versions was comparable: MPI-IO added a small overhead of at most 7–8% on top of native PVFS. We believe this overhead can be reduced further with careful tuning.

4.3 BTIO Benchmark

The BTIO benchmark [21] from NASA Ames Research Center simulates the I/O required by a time-stepping flow solver that periodically writes its solution matrix. The solution matrix is distributed among processes by using a multipartition distribution [6] in which each process is responsible for several disjoint subblocks of points (cells) of the grid. The solution matrix is stored on each process as C three-dimensional arrays, where

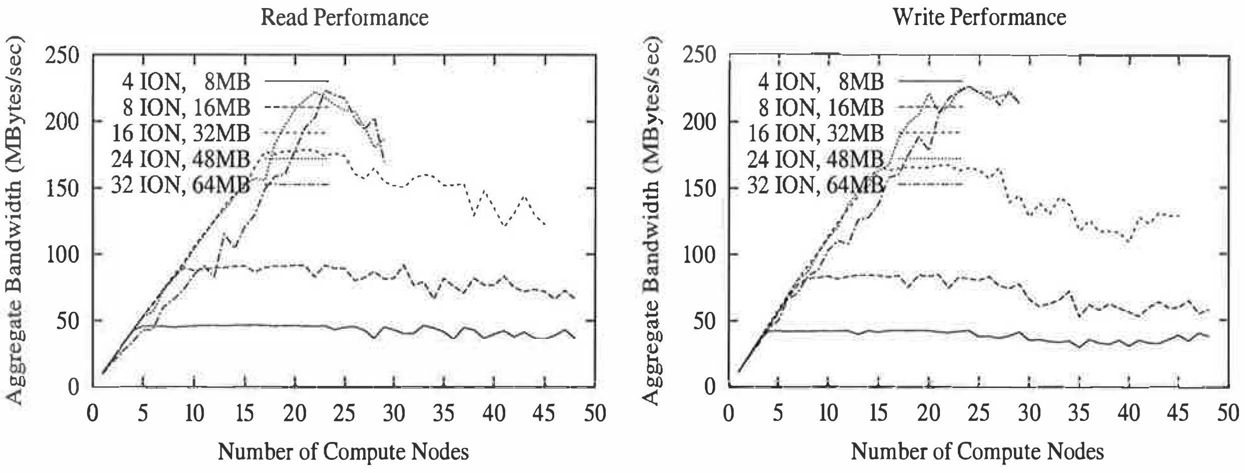


Figure 5: PVFS performance with fast ethernet

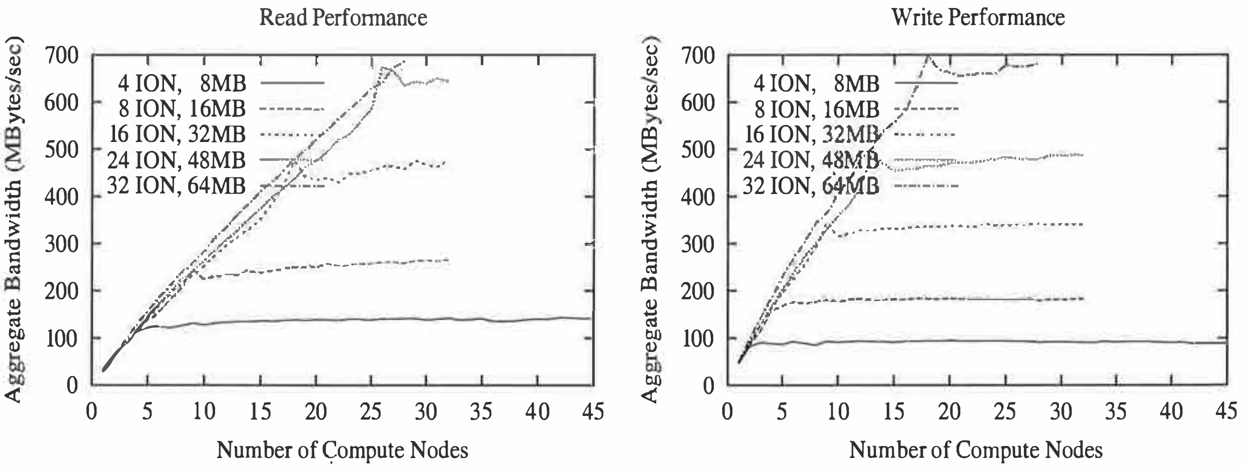


Figure 6: PVFS performance with Myrinet

C is the number of cells on each process. (The arrays are actually four dimensional, but the first dimension has only five elements and is not distributed.) Data is stored in the file in an order corresponding to a column-major ordering of the global solution matrix.

The access pattern in BTIO is noncontiguous in memory and in the file and is therefore difficult to handle efficiently with the UNIX/POSIX I/O interface. We used the "full MPI-IO" version of this benchmark, which uses MPI derived datatypes to describe noncontiguity in memory and file and uses a single collective I/O function to perform the entire I/O. The ROMIO implementation of MPI-IO optimizes such a request by merging the accesses of different processes and making large, well-formed requests to the file system [31].

The benchmark, as obtained from NASA Ames, performs only writes. In order to measure the read bandwidth for the same access pattern, we modified the benchmark to also perform reads. We ran the Class C problem size, which uses a $162 \times 162 \times 162$ element array with a total size of 162 Mbytes. The number of I/O nodes was fixed at 16, and tests were run using 16, 25, and 36 compute nodes (the benchmark requires that the number of compute nodes be a perfect square). Table 2 summarizes the results.

With fast ethernet, the maximum performance was reached with 25 compute nodes. With more compute nodes, the smaller granularity of each I/O access resulted in lower performance. For this configuration, we attained 49% of the peak concurrent-read performance and 61% of the peak concurrent-write performance mea-

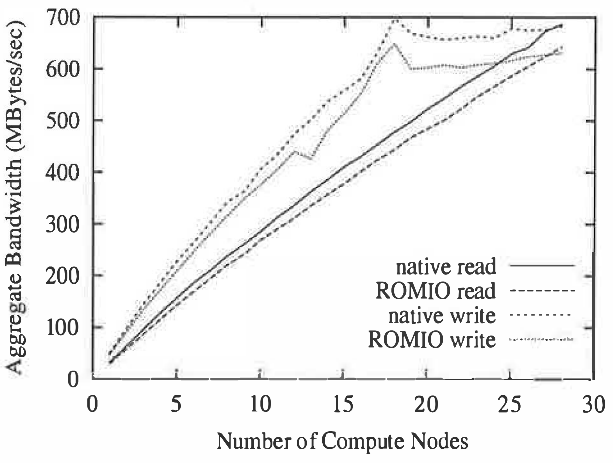


Figure 7: ROMIO versus native PVFS performance with Myrinet and 32 I/O nodes

Table 2: BTIO performance (Mbytes/sec), 16 I/O nodes, Class C problem size (162 × 162 × 162).

Compute	Fast Ethernet		Myrinet	
Nodes	read	write	read	write
16	83.8	79.1	156.7	157.3
25	88.4	101.3	197.3	192.0
36	66.3	61.1	232.3	230.7

sured in Section 4.1. The other time was spent in the computation and communication required to merge the accesses of different processes in ROMIO's collective I/O implementation. Without this merging, however, the performance would have been significantly lower because of the numerous small reads and writes in this application.

With Myrinet, the maximum performance was reached with 36 compute nodes. Here we again see the benefit of a high-speed network in that even for the smaller requests resulting from using more compute nodes, we were able to attain higher performance. The performance obtained was about 51% of the peak concurrent-read performance and 70% of peak concurrent-write performance measured in Section 4.1.

5 Conclusions and Future Work

PVFS brings high-performance parallel file systems to Linux clusters and, although more testing and tuning are needed for production use, it is ready and available for use now. The inclusion of PVFS support in the ROMIO MPI-IO implementation makes it easy for applications written portably with the MPI-IO API to take advantage of the available disk subsystems lying dormant in most Linux clusters.

PVFS also serves as a tool that enables us to pursue further research into various aspects of parallel I/O and parallel file systems for clusters. We outline some of our plans below.

One limitation of PVFS, at present, is that it uses TCP for all communication. As a result, even on fast gigabit networks, the communication performance is limited to that of TCP on those networks, which is usually unsatisfactory. We are therefore redesigning PVFS to use TCP as well as faster communication mechanisms (such as VIA [34], GM [20], and ST [25]) where available. We plan to design a small communication abstraction that captures PVFS's communication needs, implement PVFS on top of this abstraction, and implement the abstraction separately on TCP, VIA, GM, and the like. A similar approach, known as an abstract device interface, has been used successfully in MPICH [12] and ROMIO [32].

Some of the performance results in this paper, particularly the cases on fast ethernet where performance drops off, suggest that further tuning is needed. We plan to instrument the PVFS code and obtain detailed performance measurements. Based on this data, we plan to investigate whether performance can be improved by tuning some parameters in PVFS and TCP, either *a priori* or dynamically at run time.

We also plan to design a more general file-partitioning interface that can handle the noncontiguous accesses supported in MPI-IO, improve the client-server interface to better fit the expectations of kernel interfaces, design a new internal I/O-description format that is more flexible than the existing partitioning scheme, investigate adding redundancy support, and develop better scheduling algorithms for use in the I/O daemons in order to better utilize I/O and networking resources.

6 Availability

Source code, compiled binaries, documentation, and mailing-list information for PVFS are available from the PVFS web site at `http://www.parl.clemson.edu/pvfs/`.

Information and source code for the ROMIO MPI-IO implementation are available at `http://www.mcs.anl.gov/romio/`.

References

[1] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 109–126. ACM Press, December 1995.

[2] Rajesh Bordawekar, Steven Landherr, Don Capps, and Mark Davis. Experimental evaluation of the Hewlett-Packard Exemplar file system. *ACM SIGMETRICS Performance Evaluation Review*, 25(3):21–28, December 1997.

[3] Peter J. Braam. The Coda distributed file system. *Linux Journal*, #50, June 1998.

[4] Peter J. Braam, Michael Callahan, and Phil Schwan. The InterMezzo filesystem. In *Proceedings of the O'Reilly Perl Conference 3*, August 1999.

[5] Tim Bray. Bonnie file system benchmark. `http://www.textuality.com/bonnie/`.

[6] J. Bruno and P. Cappello. Implementing the Beam and Warming Method on the Hypercube. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, January 1988.

[7] Chiba City, the Argonne scalable cluster. `http://www.mcs.anl.gov/chiba/`.

[8] Coda file system. `http://www.coda.cs.cmu.edu`.

[9] Peter F. Corbett and Dror G. Feitelson. The Vesta parallel file system. *ACM Transactions on Computer Systems*, 14(3):225–264, August 1996.

[10] Peter F. Corbett, Dror G. Feitelson, Jean-Pierre Prost, George S. Almasi, Sandra Johnson Baylor, Anthony S. Bolmarcich, Yarsun Hsu, Julian Satran, Marc Snir, Robert Colao, Brian Herr, Joseph Kavaky, Thomas R. Morgan, and Anthony Zlotek. Parallel file systems for the IBM SP computers. *IBM Systems Journal*, 34(2):222–248, January 1995.

[11] Intel Scalable Systems Division. Paragon system user's guide. Order Number 312489-004, May 1995.

[12] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message-passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[13] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.

[14] Jay Huber, Christopher L. Elford, Daniel A. Reed, Andrew A. Chien, and David S. Blumenthal. PPFS: A high performance portable parallel file system. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 385–394, Barcelona, July 1995. ACM Press.

[15] IEEE/ANSI Std. 1003.1. Portable operating system interface (POSIX)–part 1: System application program interface (API) [C language], 1996 edition.

[16] InterMezzo. `http://www.inter-mezzo.org`.

[17] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical Report Computer Sciences Technical Report #1346, University of Wisconsin-Madison, April 1997.

[18] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface, July 1997. `http://www.mpi-forum.org/docs/docs.html`.

[19] Steven A. Moyer and V. S. Sunderam. PIOUS: a scalable parallel I/O system for distributed computing environments. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 71–78, 1994.

[20] Myrinet software and documentation. `http://www.myri.com/scs`.

[21] NAS application I/O (BTIO) benchmark. `http://parallel.nas.nasa.gov/MPI-IO/btio`.

[22] Nils Nieuwejaar and David Kotz. The Galley parallel file system. *Parallel Computing*, 23(4):447–476, June 1997.

[23] Kenneth W. Preslan, Andrew P. Barry, Jonathan E. Brassow, Grant M. Erickson, Erling Nygaard, Christopher J. Sabol, Steven R. Soltis, David C. Teigland, and Matthew T. O'Keefe. A 64-bit, shared disk file system for Linux. In *Proceedings of the Seventh NASA Goddard Conference on Mass Storage Systems*. IEEE Computer Society Press, March 1999.

[24] ROMIO: A High-Performance, Portable MPI-IO implementation. `http://www.mcs.anl.gov/romio`.

[25] Scheduled Transfer—application programming interface mappings (ST-API). `http://www.hippi.org/cSTAPI.html`.

[26] Huseyin Simitci, Daniel A. Reed, Ryan Fox, Mario Medina, James Oly, Nancy Tran, and Guoyi Wang. A framework for adaptive storage of input/output on computational grids. In *Proceedings of the Third Workshop on Runtime Systems for Parallel Programming*, 1999.

[27] Hal Stern. *Managing NFS and NIS*. O'Reilly & Associates, Inc., 1991.

[28] Hakan Taki and Gil Utard. MPI-IO on a parallel file system for cluster of workstations. In *Proceedings of the IEEE Computer Society International Workshop on Cluster Computing*, pages 150–157. IEEE Computer Society Press, December 1999.

[29] Test TCP. `ftp://ftp.arl.mil/pub/ttcp/`.

[30] Rajeev Thakur, William Gropp, and Ewing Lusk. An abstract-device interface for implementing portable parallel-I/O interfaces. In *Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation*, pages 180--187. IEEE Computer Society Press, October 1996.

[31] Rajeev Thakur, William Gropp, and Ewing Lusk. Data sieving and collective I/O in ROMIO. In *Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189. IEEE Computer Society Press, February 1999.

[32] Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems*, pages 23–32. ACM Press, May 1999.

[33] The parallel virtual file system. `http://www.parl.clemson.edu/pvfs/`.

[34] VI architecture. `http://www.viarch.org`.

[35] XFS: A next generation journalled 64-bit filesystem with guaranteed rate I/O. `http://www.sgi.com/Technology/xfs-whitepaper.html`.

Maximizing Beowulf Performance

Robert G. Brown
Duke University Physics Department
Box 90305, Durham, NC, 27708-0305
rgb@phy.duke.edu, http://www.phy.duke.edu/~rgb

Abstract

At this point in time the beowulf (and other related compute cluster) architectures has come of age in Linux. Few indeed are those in any realm of technical computing that are unaware of the fact that one can assemble a collection of commodity off the shelf (COTS) computers and networking hardware into a high performance supercomputing environment. However, a detailed knowledge or appreciation for the bottlenecks and special problems associated with beowulf design is not so common. A review of the important bottlenecks and design features of a beowulf is given along with associated benchmarking and measurement tools to illustrate how to bridge the gap between the simple "recipe" of a beowulf as a pile of compute nodes, interconnected with a fast network and running linux and the realities of engineering a parallel code and beowulf-style cluster to achieve satisfactory performance.

1 Introduction

A very simple recipe for a beowulf [beowulf] might be: Purchase N more or less identical COTS computers for compute nodes. Connect them by a COTS fast private network, for example switched fast ethernet. Serve them, isolate them and access them from auxiliary nodes (which may well be a single common "head node"). Install Linux and a small set of more or less standard parallel computation tools (typically PVM and MPI, along with several others that might or might not be present in a given installation). Voila! A beowulf supercomputer[1]

[1]Much of this paper applies as well to any linux based cluster, including simple LAN clusters of workstations used as nodes. Many of these designs do not form, strictly speaking, a "beowulf supercomputer", but the term "beowulf" in this paper will be used for all beowulf-style clusters.

Although this recipe is fair enough and will yield acceptable performance (measured in terms of "speedup" scaling as illustrated below) in many parallelized applications, in others it will not. In many cases the performance obtained will depend on the *details* of the node and network design as well as the design and implementation of the parallel application itself. The critical decisions made during the design process are informed by a deep and quantitative analysis of the fundamental rates and performance features of both the nodes and the network. The understanding of the important design criteria and how they correspond to features of the parallel application is a hallmark of a well-conceived beowulf project or proposal.

In the following sections we will briefly review the essential elements of successful beowulf design. They are:

- Understand Amdahl's Law and the fundamental limitations of parallel program design. The most common mistake of novices is to expect more from a parallelization of a program than is physically possible.
- Understand the fundamental rates (and latencies and bandwidths) of your computational hardware. In particular, know something about the fundamental latencies and bandwidths available in the various relevant subsystems (CPU, memory, network) and how they can act as bottlenecks to your parallelized subtasks. These rates should be known *quantitatively* to inform a sound beowulf design. Tools to use to make these measurements will be explicitly illustrated in the context of two simple beowulf nodes.

The goal of the discussion will be to convey an appreciation for some of the important design decisions to be made that is both *qualitative* and *quanti-*

tative. The use of some mathematics is unavoidable but will be explained in the simplest possible terms.

With a general understanding of the scaling of parallel computation clusters of the general beowulf design in hand, it should be straightforward to select a successful and cost effective design for any parallelizeable problem that lies within the general range of the beowulf concept. Although it isn't a strict rule, the beowulf designs we will focus on are those appropriate for solving complex mathematical or numerical problems such as those encountered in physics, statistics, weather prediction, chemistry, and many other numerical venues.

This paper will *not* address clustering for purposes of failover and reliability or load balancing in e.g. a database server or webserver context. Although linux clusters are increasingly in use in these contexts, these clusters are not beowulfs.

It will also not address the more esoteric aspects of parallel program design (not intending at all to minimize the importance of a sound program design in successful beowulf operation) and indeed the mathematical treatment presented of parallel scaling may appear naive to those familiar with the entire multidimensional theory of the various algorithms that might be used to treat a given problem. We will instead be satisfied with providing references to some of the many authoritative works that address this subject far better than we would find possible in a short paper.

In the paper below, we will begin by addressing the basic theory of speedup and scaling in parallel computation. From this we will move on to a description of the important microscopic rates and measures that determine beowulf design and performance and a discussion of tools that can be used to measure them.

2 Amdahl's Law & Parallel Speedup

The theory of doing computational work in parallel has some fundamental laws that place limits on the benefits one can derive from parallelizing a computation (or really, any kind of work). To understand these laws, we have to first define the objective. In general, the goal in large scale computation is to get as much work done as possible in the shortest possible time within our budget. We "win" when we can do a big job in less time or a bigger job in the same time and not go broke doing so. The "power" of a computational system might thus be usefully defined to be the amount of computational work that can be done divided by the time it takes to do it, and we generally wish to optimize power per unit cost, or cost-benefit.

Physics and economics conspire to limit the raw power of individual single processor systems available to do any particular piece of work even when the dollar budget is effectively unlimited. The cost-benefit scaling of increasingly powerful single processor systems is generally nonlinear and very poor – one that is twice as fast might cost four times as much, yielding only half the cost-benefit, per dollar, of a cheaper but slower system. One way to increase the power of a computational system (for problems of the appropriate sort) past the economically feasible single processor limit is to apply more than one computational engine to the problem.

This is the motivation for beowulf design and construction; in many cases a beowulf may provide access to computational power that is available in a alternative single or multiple processor designs, but only at a far greater cost.

In a perfect world, a computational job that is split up among N processors would complete in $1/N$ time, leading to an N-fold increase in power. However, any given piece of parallelized work to be done will contain parts of the work that *must* be done serially, one task after another, by a single processor. This part does *not* run any faster on a parallel collection of processors (and might even run more slowly). Only the part that can be parallelized runs as much as N-fold faster.

The "speedup" of a parallel program is defined to be the ratio of the rate at which work is done (the power) when a job is run on N processors to the rate at which it is done by just one. To simplify the discussion, we will now consider the "computational work" to be accomplished to be an arbitrary task (generally speaking, the particular problem of greatest interest to the reader). We can then define the speedup (increase in power as a function of N) in terms of the time required to complete this particular fixed piece of work on 1 to N processors.

Let $T(N)$ be the time required to complete the task on N processors. The speedup $S(N)$ is the ratio

$$S(N) = \frac{T(1)}{T(N)}. \qquad (1)$$

In many cases the time $T(1)$ has, as noted above, both a serial portion T_s and a parallelizeable portion T_p. The serial time does not diminish when the parallel part is split up. If one is "optimally" fortunate, the parallel time is decreased by a factor of $1/N$). The speedup one can expect is thus

$$S(N) = \frac{T(1)}{T(N)} = \frac{T_s + T_p}{T_s + T_p/N}. \qquad (2)$$

This elegant expression is known as *Amdahl's Law* [Amdahl] and is usually expressed as an inequality. This is in almost all cases the *best* speedup one can achieve by doing work in parallel, so the real speed up $S(N)$ is less than or equal to this quantity.

Amdahl's Law immediately eliminates many, many tasks from consideration for parallelization. If the serial fraction of the code is not much smaller than the part that could be parallelized (if we rewrote it and were fortunate in being able to split it up among nodes to complete in less time than it otherwise would), we simply won't see much speedup no matter how many nodes or how fast our communications. Even so, Amdahl's law is still far too optimistic. It ignores the overhead incurred due to parallelizing the code. We must generalize it.

A fairer (and more detailed) description of parallel speedup includes at least two more times of interest:

T_s The original single-processor serial time.

T_{is} The (average) additional *serial* time spent doing things like interprocessor communications (IPCs), setup, and so forth in all parallelized tasks. This time can depend on N in a variety of ways, but the simplest assumption is that each system has to expend this much time, one after the other, so that the total additional serial time is for example $N * T_{is}$.

T_p The original single-processor parallelizeable time.

T_{ip} The (average) *additional* time spent by each processor doing just the setup and work that it does in parallel. This may well include idle time, which is often important enough to be accounted for separately.

It is worth remarking that generally, the most important element that contributes to T_{is} is the time required for communication between the parallel subtasks. This communication time is always there – even in the simplest parallel models where identical jobs are farmed out and run in parallel on a cluster of networked computers, the remote jobs must be begun and controlled with messages passed over the network. In more complex jobs, partial results developed on each CPU may have to be sent to all other CPUs in the beowulf for the calculation to proceed, which can be *very* costly in scaled time. As we'll see below, T_{is} in particular plays an extremely important role in determining the speedup scaling of a given calculation. For this (excellent!) reason many beowulf designers and programmers are obsessed with communications hardware and algorithms.

It is common to combine T_{ip}, N and T_{is} into a single expression $T_o(N)$ (the "overhead time") which includes any complicated N-scaling of the IPC, setup, idle, and other times associated with the overhead of running the calculation in parallel, as well as the scaling of these quantities with respect to the "size" of the task being accomplished. The description above (which we retain as it illustrates the generic form of the relevant scalings) is still a *simplified* description of the times – real life parallel tasks can be much more complicated, although in many cases the description above is adequate.

Using these definitions and doing a bit of algebra, it is easy to show that an improved (but still simple) estimate for the parallel speedup resulting from splitting a particular job up between N nodes (assuming one processor per node) is:

$$S(N) = \frac{T_s + T_p}{T_s + N * T_{is} + T_p/N + T_{ip}}. \qquad (3)$$

This expression will suffice to get at least a general feel for the scaling properties of a task that might be parallelized on a typical beowulf.

It is useful to plot the dimensionless "real-world speedup" (3) for various *relative* values of the times. In all the figures below, $T_s = 10$ (which sets our basic scale, if you like) and $T_p = 10, 100, 1000, 10000, 100000$ (to show the systematic effects of parallelizing more and more work compared to T_s).

The primary determinant of beowulf scaling performance is the amount of (serial) work that must be done to set up jobs on the nodes and then in com-

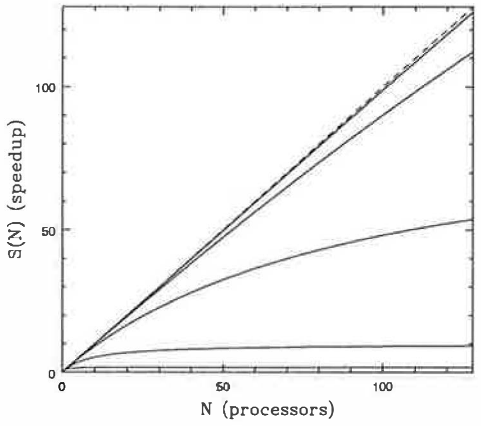


Figure 1: $T_{is} = 0$ and $T_p = 10, 100, 1000, 10000, 100000$ (in increasing order).

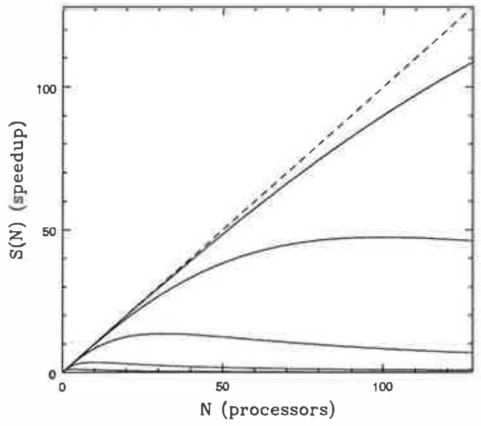


Figure 2: $T_{is} = 10$ and $T_p = 10, 100, 1000, 10000, 100000$ (in increasing order).

munications between the nodes, the time that is represented as T_{is}. All figures have $T_{ip} = 1$ fixed; this parameter is rather boring as it effectively adds to T_s and is often very small.

Figure 1 shows the kind of scaling one sees when communication times are negligible compared to computation. This set of curves is roughly what one expects from Amdahl's Law alone, which was derived with no consideration of IPC overhead. Note that the dashed line in all figures is perfectly linear speedup, which is never obtained over the entire range of N although one can often come close for small N or large enough T_p.

In figure 2, we show a fairly typical curve for a "real" beowulf, with a relatively small IPC overhead of $T_{is} = 1$. In this figure one can see the advantage of cranking up the parallel fraction (T_p relative to T_s) and can also see how even a relatively small serial communications process on each node causes the gain curves to peak well short of the saturation predicted by Amdahl's Law in the first figure. Adding processors past this point *costs* one speedup. Increasing T_{is} further (relative to everything else) causes the speedup curves to peak earlier and at smaller values.

Finally, in figure 3 we continue to set $T_{is} = 1$, but this time with a *quadratic* N dependence $N^2 * T_{is}$ of the serial IPC time. This might result if the communications required between processors is long range (so every processor must speak to every other processor) and is not efficiently managed by a suitable algorithm. There are other ways to get nonlinear dependences of the additional serial time on N, and as this figure clearly shows they can have a profound effect on the per-processor scaling of the speedup.

As one can clearly see, unless the ratio of T_p to T_{is} is in the ballpark of 100,000 to 1 one cannot actually *benefit* from having 128 processors in a "typical" beowulf. At only 10,000 to 1, the speedup saturates at around 100 processors and then decreases. When the ratio is even smaller, the speedup peaks with only a handful of nodes working on the problem. From this we learn some important lessons. The most important one is that for many problems simply adding processors to a beowulf design won't provide any additional speedup and could even slow a calculation down *unless one also scales up the problem* (increasing the T_p to T_{is} ratio) as well.

The scaling of a given calculation has a significant impact on beowulf engineering. Because of overhead, speedup is not a matter of just adding the speed of however many nodes one applies to a given problem. For some problems it is clearly advantageous to trade off the *number* of nodes one purchases (for example in a problem with small T_s and $T_p/T_{is} \approx 100$) in order to purchase tenfold improved communications (and perhaps alter the T_p/T_{is} ratio to 1000).

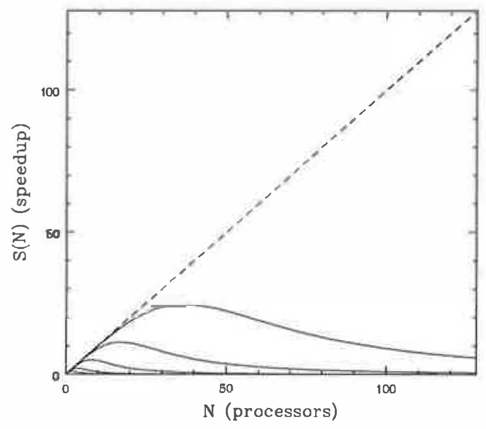


Figure 3: $T_{is} = 10$ and $T_p = 10, 100, 1000, 10000, 100000$ (in increasing order) with T_{is} contributing *quadratically* in N.

The nonlinearities prevent one from developing any simple rules of thumb in beowulf design. There are times when one obtains the greatest benefit by selecting the fastest possible processors and network (which reduce both T_s and T_p in absolute terms) instead of buying more nodes because we know that the rate equation above will limit the parallel speedup we might ever hope to get even with the fastest nodes. Paradoxically, there are other times that we can do better (get better speedup scaling, at any rate) by buying *slower* processors (when we are network bound, for example), as this can also increase T_p/T_{is}. In general, one should be aware of the peaks that occur at the various scales and not naively distribute small calculations (with small T_p/T_{is}) over more processors than they can use.

In summary, parallel performance depends primarily on certain relatively simple parameters like T_s, T_p and T_{is} (although there may well be a devil in the details that we've passed over). These parameters, in turn are at least partially under our control in the form of programming decisions and hardware design decisions. Unfortunately, they depend on many microscopic measures of system and network performance that are inaccessible to many potential beowulf designers and users. T_p clearly should depend on the "speed" of a node, but the single node speed itself may depend *nonlinearly* on the speed of the processor, the size and structure of the caches, the operating system, and more.

Because of the nonlinear complexity, there is no way to *a priori* estimate expected performance on the basis of any simple measure. There is still considerable benefit to be derived from having in hand a set of quantitative measures of "microscopic" system performance and gradually coming to understand how one's program depends on the potential bottlenecks they reveal. The remainder of this paper is dedicated to reviewing the results of applying a suite of microbenchmark tools to a pair of nodes to provide a quantitative basis for further beowulf hardware and software engineering.

3 Microbenchmarking Tools

From the previous section we can see that there are several things that we have to understand fairly thoroughly to design a beowulf to cost-effectively tackle a given problem. To achieve the best scaling behavior, we want to maximize the parallel fraction of a program (the part that can be split up) and minimize the serial fraction (which cannot). We also want to maximize the time spend doing work in parallel on each node and minimize the time required to communicate between nodes. We want to avoid wasting time by having some nodes sit idle waiting for other nodes to finish.

However, we must be cautious and clearly define our real goals as in general they aren't to "achieve the best scaling behavior" (unless one is a computer scientist studying the abstract problem, of course). More commonly in application, they are to "get the most work done in the least amount of time given a fixed budget". When *economic constraints* appear in the picture one has to carefully consider tradeoffs between the computational speed of the nodes, the speed and latency of the network, the size of memory and its speed and latency, and the size, speed and latency of any hard storage subsystem that might be required. Virtually any conceivable combination of system and network speed can turn out to be cost-benefit optimal and get the most work done for a given budget and parallel task.

Finding the truly optimum design can be somewhat difficult. In some cases the *only* way to determine a program's performance on a given hardware and software platform (or beowulf design) is to do a lot of prototyping and determine the best design empirically (where hopefully one has enough funding in these cases to fund the prototyping and then scale the successful design up into the production beowulf). This is almost always the *best* thing to do, if one can afford it. In all cases, the design process is significantly easier if one possesses a detailed and quantitative knowledge of various microscopic *rates, latencies,* and *bandwidths.*

- A *rate* is a given number of operations per unit time, for example, the number of double precision multiplications a CPU can execute per second. We might like to know the "maximum" rate a CPU can execute floating point instructions under ideal circumstances. We might be even more interested in how the "real world" floating point rate depends on (for example) the size and locality of the memory references being operated upon.
- A *latency* is is the time the CPU (or other subsystem) has to *wait* for a resource or service to become available after it is requested and has units of an inverse rate – milliseconds per disk seek, for example. A latency isn't necessarily the inverse of a rate, however, because the latency often is very different for an isolated request and a streaming series of identical requests.
- A *bandwidth* is a special case of a rate. It measures "information per unit time" being delivered between subsystems (for example between memory and the CPU). Information in the context of computers is typically data or code organized as a byte stream, so a typical unit of bandwidth might be megabytes per second.

Latency is *very* important to understand and quantify as in many cases our nodes will be literally sitting there and twiddling their thumbs waiting for a resource. Latencies may be the *dominant* contribution to the communications times in our performance equations above. Also (as noted) rates are often the inverse of some latency. One can equally well talk about the rate that a CPU executes floating point instructions or the latency (the time) between successive instructions which is its inverse. In other cases such as the network, memory, or disk, latency is just one factor that contributes to overall rates of streaming data transfer. In general a large latency translates into a low rate (for the same resource) for a small or isolated request.

Clearly these rates, latencies and bandwidths are important determinants of program performance even for single threaded programs running on a single computer. Taking advantage of the nonlinearities (or avoiding their *dis*advantages can result in dramatic improvements in performance, as the ATLAS (Automatically Tuned Linear Algebra System) [ATLAS] project has recently made clear. By adjusting both algorithm and blocksize to maximally exploit the empirical speed characteristics of the CPU in interaction with the various memory subsystems, ATLAS achieves a factor of two or more improvement in the excution speed of a number of common linear operations. Intelligent and integrated beowulf design can similarly produce startling improvements in both cost-benefit and raw performance for certain tasks.

It would be very useful to have automatically available all of the basic rates that might be useful for automatically tuning program and beowulf design. At this time there is no daemon or kernel module that can provide this empirically determined and standardized information to a compiled library. As a consequence, the ATLAS library build (which must measure the key parameters in place) is so complex that it can take hours to build on a fast system.

There do exist various standalone (open source) microbenchmarking tools that measure a large number of the things one might need to measure to guide thoughtful design. Unfortunately, many of these tools measure only isolated performance characteristics, and as we will see below, isolated numbers are not always useful. However, one toolset has emerged that by design contains (or will soon contain) a ***full suite*** of the elementary tools for measuring precisely the rates, latencies, and bandwidths that we are most interested in, using a common and thoroughly tested timing harness. This tool is not complete[2] but it has the promise of becoming *the* fundamental toolset to support systems engineering and cluster design. It is Larry McVoy and Carl Staelin's "lmbench" toolset[lmbench].

There are two areas where the alpha version 2 of this toolset used in this paper was still missing tools to measure network throughput and raw "numerical" CPU performance (although many of the missing features and more have recently been added to lmbench by Carl Staelin after some gentle pestering). The well-known netperf (version 2.1, patch level 3)

[2]More time was spent by the author of this paper working on and with the tool than on the paper:-)

[netperf] and a privately written tool [cpu-rate] were used for this in the meantime.

All of the tools that will be discussed are open source in the sense that their source can be readily obtained on the network and that no royalties are charged for its use. The lmbench suite, however, has a general use license that is slightly more restricted than the usual Gnu Public License (GPL) as described below.

In the next subsections the results of applying these tools to measure system performance in my small personal beowulf cluster[Eden] will be presented. This cluster is moderately heterogeneous and functions in part as a laboratory for beowulf development. A startlingly complete and clear profile of system performance and its dependence on things like code size and structure will emerge.

3.1 Lmbench Results

In order to ***publish*** lmbench results in a public forum, the lmbench license ***requires*** that the benchmark code must be compiled with a "standard" level of optimization (-O only) and that *all* the results produced by the lmbench suite must be published. These two rules together ensure that the results produced compare as fairly as possible apples to apples when considering multiple platforms, and prevents vendors or overzealous computer scientists from seeking "magic" combinations of optimizations that improve one result (which they then selectively publish) at the expense of others.

Accordingly, on the following page is a full set of lmbench results generated for "lucifer", the primary server node for my home (primarily development) beowulf [Eden]. The mean values and error estimates were generated from averaging ten independent runs of the full benchmark. lucifer is a 466 MHz *dual* Celeron system, permitting it to function (in principle) simultaneously as a master node and as a participant node. The cpu-rate results are also included on this page for completeness although they may be superseded by Carl Staelin's superior hardware instruction latency measures in the future.

lmbench clearly produces an *extremely detailed* picture of microscopic systems performance. Many of these numbers are of obvious interest to beowulf designers and have indeed been discussed (in many cases without a sound quantitative basis) on the be-

HOST	lucifer
CPU	Celeron (Mendocino) (x2)
CPU Family	i686
MHz	467
L1 Cache Size	16 KB (code)/16 KB (data)
L2 Cache Size	128 KB
Motherboard	Abit BP6
Memory	128 MB of PC100 SDRAM
OS Kernel	Linux 2.2.14-5.0smp
Network (100BT)	Lite-On 82c168 PNIC rev 32
Network Switch	Netgear FS108

Table 1: Lucifer System Description

null call	0.696 ± 0.006
null I/O	1.110 ± 0.005
stat	3.794 ± 0.032
open/close	5.547 ± 0.054
select	44.7 ± 0.82
signal install	1.971 ± 0.006
signal catch	3.981 ± 0.002
fork proc	634.4 ± 28.82
exec proc	2755.5 ± 10.34
shell proc	10569.0 ± 46.92

Table 2: lmbench latencies for selected processor/process activities. The values are all times in microseconds averaged over ten independent runs (with error estimates provided by an unbiased standard deviation), so "smaller is better".

2p/0K	1.91 ± 0.036
2p/16K	14.12 ± 0.724
2p/64K	144.67 ± 9.868
8p/0K	3.30 ± 1.224
8p/16K	48.45 ± 1.224
8p/64K	201.23 ± 2.486
16p/0K	6.26 ± 0.159
16p/16K	63.66 ± 0.779
16p/64K	211.38 ± 5.567

Table 3: Lmbench latencies for context switches, in microseconds (smaller is better).

pipe	10.62 ± 0.069
AF UNIX	33.74 ± 3.398
UDP	55.13 ± 3.080
TCP	127.71 ± 5.428
TCP Connect	265.44 ± 7.372
RPC/UDP	140.06 ± 7.220
RPC/TCP	185.30 ± 7.936

Table 4: Lmbench *local* communication latencies, in microseconds (smaller is better).

UDP	164.91 ± 2.787
TCP	187.92 ± 9.357
TCP Connect	312.19 ± 3.587
RPC/UDP	210.65 ± 3.021
RPC/TCP	257.44 ± 4.828

Table 5: Lmbench *network* communication latencies, in microseconds (smaller is better).

L1 Cache	6.00 ± 0.000
L2 Cache	112.40 ± 7.618
Main mem	187.10 ± 1.312

Table 6: Lmbench *memory* latencies in nanoseconds (smaller is better). Also see graphs for more complete picture.

pipe	290.17 ± 11.881
AF UNIX	64.44 ± 3.133
TCP	31.70 ± 0.663
UDP	(not available)
bcopy (libc)	79.51 ± 0.782
bcopy (hand)	72.93 ± 0.617
mem read	302.79 ± 3.054
mem write	97.92 ± 0.787

Table 7: Lmbench *local* communication bandwidths, in 10^6 bytes/second (bigger is better).

TCP	11.21 ± 0.018
UDP	(not available)

Table 8: Lmbench *network* communication bandwidths, in 10^6 bytes/second (bigger is better).

Single precision	289.10 ± 1.394
Double precision	299.09 ± 2.295

Table 9: CPU-rates in BOGOMFLOPS – 10^6 simple arithmetic operations/second, in L1 cache (bigger is better). Also see graph for out-of-cache performance.

owulf list [beowulf]. We must focus in order to conduct a sane discussion in the allotted space. In the following subsections on we will consider the network, the memory, and the cpu-rates as primary contributors to beowulf and parallel code design.

These are not at all independent. The rate at which the system does floating point arithmetic on streaming vectors of numbers is very strongly determined by the relative size of the L1 and L2 cache and the size of the vector(s) in question. Significant (and somewhat unexpected) structure is also revealed in network performance as a function of packet size, which suggests "interesting" interactions between the network, the memory subsystem, and the operating system that are worthy of further study.

3.2 Netperf Results

Netperf is a venerable and well-written tool for measuring a variety of critical measures of network performance. Some of its features are still not duplicated in the lmbench 2 suite; in particular the ability to completely control variables such as overall message block size and packet payload size.

A naive use of netperf might be to just call

```
netperf -H targethost
```

to get a quick and dirty measurement of TCP stream bandwidth to a given target. However, as the lmbench TCP latency shows (see table 5), it takes some 150-200 microseconds to transmit a one-byte TCP packet message (on lucifer) or at most 5000-7000 packets can be sent per second. For small packets this results in far less than the "wirespeed dominated" bandwidth – the actual bandwidth observed for small messages is dominated by *latency*.

For each message sent, the time required goes directly into an IPC time like T_{is}. In the minimum 200 microseconds that are lost, the CPU could have done tens of thousands of floating point operations! This is why network latency is an extremely important parameter in beowulf design.

Bandwidth is also important – sometimes one has only a single message to send between processors, but it is a large one and takes much more than the 200 microseconds latency penalty to send. As message sizes get bigger the system uses more and more of the *total* available bandwidth and is less affected by latency. Eventually throughput saturates

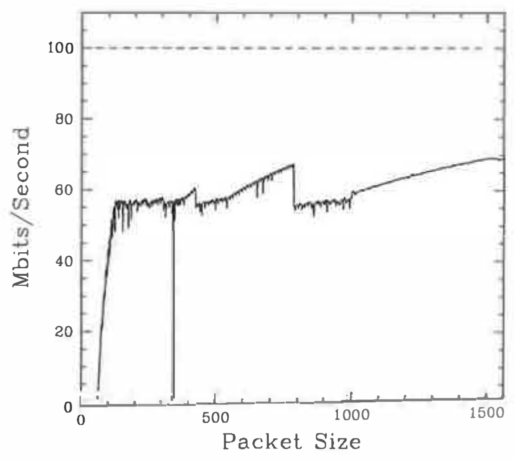


Figure 4: TCP Stream (netperf) measurements of bandwidth as a function of packet size between lucifer and eve.

at some maximum value that depends on many variables. Rather than try to understand them all, it is is easier (and more accurate) to determine (maximum) bandwidth as a function of message size by direct measurement.

Both netperf and bw_tcp in lmbench allow one to directly select the message size (in bytes) to make a measurements of streaming TCP throughput. With a simple perl script one can generate a fine-grained plot of overall performance as a function of packet size. This has been done for a 100BT connection between lucifer and "eve" (a reasonably similar host on the same switch) as a function of packet size. These results are shown in figure 4.

Figure 4 reveals a number of surprising and even disappointing features. Bandwidth starts out small at a message size of one byte (and a packet size of 64 bytes, including the header) and rapidly grows roughly linearly at first as one expects in the latency-dominated regime where the number of packets per second is constant but the size of the packets is increasing. However, the bandwidth appears to *discontinuously* saturate at around 55 Mbps for packet sizes around 130 bytes long or longer. There is also considerable (unexpected) structure even in the saturation regime with sharp packet size thresholds. The same sort of behavior (with somewhat different structure and a bit better asymptotic large packet performance) appears when bw_tcp is used to perform the same measurement. We see that the single lmbench result of a somewhat low but relatively normal 11.2 MBps (90 Mbps) for large packets in table 8 hides a wealth of detail and potential IPC problems, although this single measure is all that would typically be published to someone seeking to build a beowulf using a given card and switch combination.

3.3 CPU Results

The CPU numerical performance is one of the most difficult components to precisely quantify. On the one hand, peak numerical performance is a measure always published by the vendor. On the other hand, this peak is basically never seen in practice and is routinely discounted.

CPU performance is known to be heavily dependent on just what the CPU does, the order in which it does it, the size and structure and bandwidths and latencies of its various memory subsystems including L1 and L2 caches, and the way the operating system manages cached pages. This dependence is extremely complex and studying one measure of performance for a particular set of parameters is not very illuminating if not misleading. In order to get any kind of feel at all for real world numerical performance, floating point instruction rates have to be determined for whole sweeps of e.g. accessed vector memory lengths.

What this boils down to is that there is very little numerical code that is truly "typical" and that it can be quite difficult to assign a single rate to floating point operations like addition, subtraction, multiplication, and division that might not be off by a factor of five or more relative to the rate that these operations are performed in *your* code. This translates into large uncertainties and variability of, for example, T_p with parallel program scale and design.

Still, it is unquestionably true that a detailed knowledge of the "MFLOPS" (millions of floating point operation per second) that can be performed in an inner loop of a calculation is important to code and beowulf design. Because of the high dimensionality of the variables upon which the rate depends (and the fact that we perforce must project onto a subspace of those variables to get any kind of performance picture at all) the resulting rate is somewhat

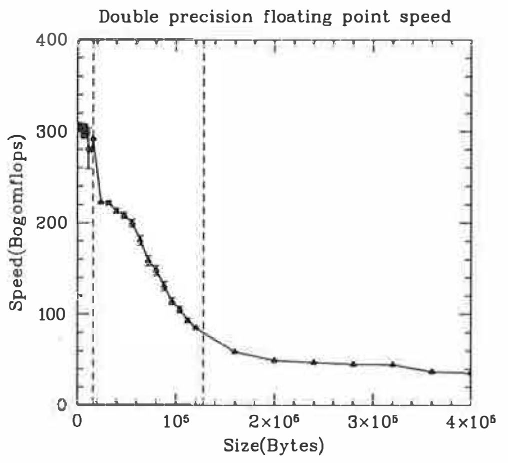


Figure 5: Double precision floating point operations per second as a function of vector length (in bytes). All points average 100 independent runs. The dashed lines indicate the locations of the L1 and L2 cache boundaries.

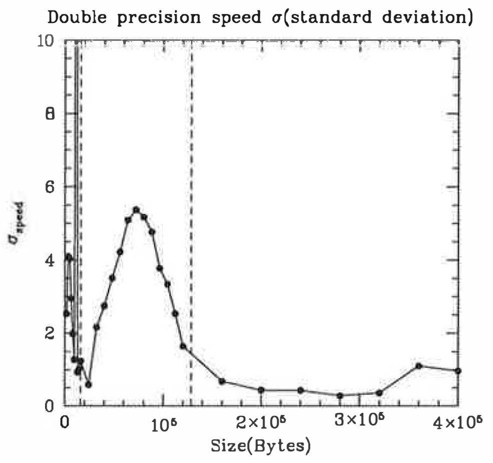


Figure 6: The standard deviation (error) associated with figure 5.

bogus but not without it uses, ***provided*** that the tool used to generate it permits the exploration of at least a few of the relevant dimensions that affect numerical performance. Perhaps the most important of these are the various memory subsystems.

To explore raw numerical performance the cpu-rate benchmark is used [cpu-rate]. This benchmark times a simple arithmetic loop over a vector of a given input length, correcting for the time required to execute the empty loop alone. The operations it executes are:

```
  x[i] = (1.0 + x[i])*(1.5 - x[i])/x[i];
```

where x[i] is initialized to be 1.0 and should end up equal to 1.0 (within any system roundoff error) afterwards as well.

Each execution of this line counts as "four floating point operations" (one of each type, where x[i] might be single or double precision) and by counting and timing one can convert this into FLOPS. As noted, the FLOPS it returns are somewhat bogus – they average over all four arithmetic operations (which may have very different rates), they contain a small amount of addressing arithmetic (to access the x[i] in the first place) that is ignored, they execute in a given order which may or many not accidentally benefit from floating point instruction pipelining in a given CPU, they presuming streaming access of the operational vector.

Still, this is more or less what what I think "most people" would mean when they ask how fast a system can do floating point arithmetic in the context of a loop over a vector. We'll remind ourselves that the results are bogus by labeling them "BOGOflops".

These rates will be ***largest*** when both the loop itself and the data it is working on are already "on the CPU" in registers, but for most practical purposes this rarely occurs in a core loop in compiled code that isn't hand built and tuned. The fastest rates one is likely to see in real life occur when the data (and hopefully the code) live in L1 cache, just outside the CPU registers. lmbench contains tests which determine at least the size of the L1 data cache size and its latency. In the case of lucifer, the L1 size is known to be 16 KB and its latency is found by lmbench to be 6 nanoseconds (or roughly 2-3 CPU clocks).

However, compiled code will ***rarely*** will fit into such a small cache unless it is specially written to do so. In any event we'd like to see what happens to the floating point speed as the length of the x[i] vector is systematically increased. Note that this measurement ***combines*** the raw numerical rate on the CPU with the effective rate that results when accounting for all the various latencies and bandwidths of the memory subsystem. Such a sequence of speeds as a function of vector lengths is graphed in figure 5.

This figure clearly shows that double precision floating point rates vary by *almost an order of magnitude* as the vector being operated on stretches from wholly within the L1 cache to several times the size of the L2 cache. Note also that the access pattern associated with the vector arithmetic is the *most favorable* one for efficient cache operation – sequential access of each vector element in turn. The factor of about *seven* difference in the execution speeds as the size of this vector is varied has profound implications for both serial code design and parallel code design. For example, the whole purpose of the ATLAS project [ATLAS] is to exploit the tremendous speed differential revealed in the figure by optimally blocking problems into in-cache loops when doing linear algebra operations numerically.

There is one more interesting feature that can be observed in this result. Because linux on Intel lacks page coloring, there is a large variability of numerical speeds observed between runs at a given vector size depending on just what pages happen to be loaded into cache when the run begins. In figure 6 the *variability* (standard deviation) of a large number (100) of independent runs of the cpu-rate benchmark is plotted as a function of vector size. One can easily pick out the the L1 and L2 cache boundaries as they neatly bracket the smooth peak visible in this figure. Although the L1 cache boundary is simple to determine directly from tests of memory speed, the L2 cache boundary has proven difficult to directly observe in benchmarks *because* of this variability. This is a new and somewhat exciting result – L2 boundaries can be revealed by a "susceptibility" of the underlying rate.

4 Conclusions

We now have many of the ingredients needed to determine how well or poorly lucifer (and its similar single-Celeron nodes, adam, eve, and abel) might perform on a simple parallel task. We also have a wealth of information to help us tune the task on *each* host to both balance the loads and to take optimal advantage of various system performance determinants such as the L1 and L2 cache boundaries and the relatively poor (or at least inconsistent) network. These numbers, along with a certain amount of judicious task profiling (for a description of the use of profiling in parallelizing a beowulf application see [profiling]) can in turn be used to determine the parameters that describe a given task like T_s, T_p, T_{is} and T_{ip}.

In addition, we have scaling curves that indicate the kind of parallel speedup we can expect to obtain for the task on the hardware we've microbenchmark-measured, and by comparing the appropriate microbenchmark numbers we *might* even be able to make a reliable guess at what the numbers and scaling would be on related but slightly different hardware (for example on a 300 MHz Celeron node instead of a 466 MHz Celeron node).

With these tools and the results they return, one can at least imagine being able to scientifically:

- develop a parallel program to run efficiently on a given beowulf
- tune an existing program on a given beowulf by considering for example bottlenecks and program scale
- develop a beowulf to run a given parallel program efficiently
- tune an existing beowulf to yield improved performance on a given program, or
- simultaneously develop, improve, and tune a *matched* beowulf design and parallel program together

even if one isn't initially a true expert in beowulf or general systems performance tuning. Furthermore, by using the *same* tools across a wide range of candidate platforms and publishing the comparative results, it may eventually become possible to do the all important optimization of *cost-benefit* that is really the fundamental motivation for using a beowulf design in the first place.

It is the hope of the author that in the near future the lmbench suite develops into a more or less standard microbenchmarking tool that can be used, along with a basic knowledge of parallel scaling theory, to identify and aggressively attack the critical bottlenecks that all too often appear in beowulf design and operation. An additional, equally interesting possibility would be to transform it into a daemon or kernel module that periodically runs on all systems and provides a standard matrix of performance measurements available from simple systems calls or via a /proc structure. This, in turn, would facilitate many, many aspects of the job of *dynamically* maximizing beowulf or general systems performance in the spirit of ATLAS but without the need to rebuild a program.

5 Acknowledgments

I would like to gratefully acknowledge the support of Duke University, the Army Research Office, Intel Corporation, who variously funded the author and/or one of his beowulfs. I am very grateful to this paper's "shepherd", Walter B. Ligon III, for reading the various drafts and making useful suggestions, and to both Larry McVoy and Carl Staelin for at least listening to my passionate plea for an unrestricted GPL for the lmbench suite and for adding a number of beowulf friendly measures. Finally, I would also especially like to thank the members and regular participants on the beowulf list.

6 Availability

All software discussed in this paper is open source and readily available over the network at the URL's indicated in the bibliography below or by sending email to the author at rgb@phy.duke.edu. The particular kind of licensing for each (GPL or not) is indicated in the reference.

References

[beowulf] See `http://www.beowulf.org` and links thereupon for a full description of the beowulf project, access to the beowulf mailing list, and more.

[Amdahl] Amdahl's law was first formulated by Gene Amdahl (working for IBM at the time) in 1967. It (and many other details of interest to a beowulf designer or parallel program designer) is discussed in detail in the following three works, among many others.

[Amalsi] G. S. Amalsi and A. Gottlieb, *Highly Parallel Computing* (2nd edition), Benjamin/Cummings, 1994.

[Foster] I. Foster, *Designing and Building Parallel Programs*, Addison-Wesley, 1995. Also see the online version of the book at Argonne National Labs, `http://www-unix.mcs.anl.gov/dbpp/`.

[Kumar] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing, Design and Analysis of Algorithms*, Benjamin/Cummings, 1994.

[lmbench] A microbenchmark toolset developed by Larry McVoy and Carl Staelin of Bitmover, Incorporated. GPL plus special restrictions. See `http://www.bitmover.com/lmbench/`.

[netperf] A network performance microbenchmark suite developed under the auspices of the Hewlett-Packard company. It was written by a number of people, starting with Rick Jones. Non-GPL open source license. See `http://www.netperf.org/`.

[cpu-rate] A crude tool for measuring "bogomflops" written by Robert G. Brown and adapted for this paper. GPL. See `http://www.phy.duke.edu/brahma`.

[Eden] The "Eden" beowulf consists of lucifer, abel, adam, eve, and sometimes caine and lilith. It lives in my home office and is used for prototyping and development.

[profiling] Robert G. Brown, *The Beowulf Design: COTS Parallel Clusters and Supercomputers*, tutorial presented for the Extreme Linux Track at the 1999 Linux Expo in Raleigh, NC. Linked to `http://www.phy.duke.edu/brahma`, along with several other introductory papers and tools of interest to beowulf developers.

[ATLAS] Automatically Tuned Linear Algebra Systems, developed by Jack Dongarra, et. al. at the Innovative Computing Laboratory of the University of Tennessee. Non-GPL open source license. See `http://www.netlib.org/atlas`.

Software Distributed Shared Memory over Virtual Interface Architecture: Implementation and Performance

Muralidharan Rangarajan* and Liviu Iftode
Department of Computer Science
Rutgers University
Piscataway, NJ 08854-8019
{muralir,iftode}@cs.rutgers.edu

Abstract

In this paper, we describe an implementation of software Distributed Shared Memory (DSM) over Virtual Interface Architecture (VIA) for a Linux-based cluster of PCs and evaluate its performance. VIA is a user-level memory-mapped communication model that provides zero-copy communication and low-overhead by excluding the operating system kernel from the communication path. To our best knowledge, our implementation is the first software DSM protocol on VIA.

The DSM protocol we have implemented on VIA is Home-based Lazy Release Consistency (HLRC) that previous studies have shown to exhibit good scalability by reducing the number of messages and memory overhead compared to the homeless counterpart. The experimental results obtained on seven Splash-2 applications show that VIA can be successfully used to support software shared memory on clusters of PCs. The paper is accompanied by a source-code distribution of the software DSM protocol for Linux/VIA clusters.

1 Introduction

System Area Networks (SANs) have become an increasingly popular solution to build scalable computer clusters by providing low latency and high bandwidth communication. Traditional communication models were unable to fully exploit the raw performance of the networks due to the high overhead added by the software protocols.

Virtual Interface Architecture (VIA) [5] is a user-level memory-mapped communication model for SANs, that reduces communication overhead by excluding the operating system kernel from the communication path. VIA is an industrial standard inspired from previous research in user-level communication performed in universities [9, 11, 10, 25]. The basic idea in user-level communication is to factor out the operating system from the critical path of communication operations. To provide protected communication, two conditions must be satisfied. First, the kernel must grant the permission for a process to communicate with another process by providing a communication channel. Second, the network interface must multiplex user-level DMA performed through these channels. This support eliminates the need to trap into the kernel each time a send is executed, and makes the send operation low-overhead. At the same time, by sending data from user space to a remote receive buffer, no copy is necessary and the end-to-end communication bandwidth will be close to the raw bandwidth provided by the network hardware.

There are multiple hardware and software implementations of VIA today. Giganet[12] has a hardware VIA implementation with drivers for Linux and Windows-NT. Firmware implementations of VIA are available for ServerNet[27] and Myrinet[22] interconnects. M-VIA [23] provides Linux software VIA drivers for various fast Ethernet cards.

The efficiency of memory-mapped communication provided by VIA doesn't come for free. As various projects started to use VIA or other memory-mapped communication libraries, it became obvious that the lack of buffer management, flow control and message packaging can make communication programming more complicated. The solution is to build high-level communication abstractions on top of VIA, while preserving its performance bene-

*Supported in part by a Student Research Grant from USENIX

fits. Recently, several message passing libraries over VIA, such as MPI [24], have been announced.

In this paper, we describe an implementation of software distributed shared memory (DSM) over VIA, for a Linux-based cluster of PCs. Software DSM [17] is available to applications as a runtime library that provides the abstraction of a shared address space across the cluster using message passing and virtual memory page protection. Given its low latency and overhead, as well as its capability to DMA directly into user address space of remote memory without intermediate copies, VIA appears very promising for software DSM. To our best knowledge, ours is the first implementation of a software DSM protocol on VIA.

The protocol we have implemented on VIA is home-based lazy release consistency (HLRC) [35, 17]. Previous studies have shown that HLRC provides good scalability by reducing the number of messages and memory overhead compared to the homeless counterpart [35]. Home-based protocols have been previously implemented on other memory-mapped interconnected clusters both for clusters of uniprocessors [20, 15] as well as for clusters of symmetric multiprocessors (SMPs) [29, 26]. Although the communication model of these networks are similar to VIA, there are a number of significant differences. For instance, compared to the virtual memory-mapped communication (VMMC) implementation on Myrinet [9], VIA requires memory registration both for send and receive, has receive queues that can be combined into completion queues (on which threads can block on explicit receive). Compared to Memory Channel [13] used in [29], VIA has no broadcast support and no implicit global ordering.

Our goal is to implement a highly efficient home-based DSM protocol exploiting the features of the VIA model and investigate its overall performance as well as the performance impact of various VIA features. For the performance evaluation we used a set of seven Splash-2 applications [33] and a cluster of eight PCs connected by Giganet VIA-based cLAN network and running Linux version 2.2.10. We were able to obtain a speedup of greater than 6 for five applications. The performance we obtained is comparable to those previously reported for home-based protocols on Myrinet/VMMC connected clusters. We have learned from our performance study that even though VIA lacks features desirable for software DSM systems, like scatter-gather and broadcast support, the VIA primitives are a good match for the requirements of the software DSM communication model.

2 Virtual Interface Architecture

The VI Architecture [5] is a user-level memory-mapped communication architecture that is designed to achieve low latency, high bandwidth across a cluster of computers. The VI architecture attempts to reduce the amount of software overhead imposed by traditional communication models, by avoiding the kernel involvement in each communication operation. In traditional models, the operating system multiplexes access to the hardware between communication endpoints and therefore all communication operations require a trap into the kernel.

Each consumer process (VI Consumer) is provided a directly accessible interface to the network hardware, called the Virtual Interface (VI). Each VI represents a communication endpoint and pairs of VIs can be connected to form communication channels for bidirectional point-to-point data transfer. Each VI has a pair of work queues, one for send and one for receive. VI Consumers send and receive messages by posting requests, in the form of descriptors, to these queues. These requests are asynchronously processed directly by network interface controller (VI Provider) and marked with a status value when completed. VI Consumers can then remove these descriptors from the queue and reuse them if necessary. Completion queues allow the VI Consumer to combine the descriptor completion events of multiple VIs into a single queue.

There are several key features of the VIA communication model:

- Direct Access to the Network Interface. This enables low latency communication which has been shown to improve DSM performance.
- Memory Registration. VIA requires that memory used for every data transfer request be registered. Any memory page registered with VIA is kept pinned to the same physical memory location until the memory is deregistered by the VI Consumer. The necessity of memory registration becomes an issue for software DSM when the shared address space is larger than the physical memory or when memory pressure

due to other applications makes it difficult to register the entire shared address space.

- Zero-Copy Protocols. With memory registration, the VI Provider can transfer data directly between the buffers of a VI Consumer and the network without copying any data to or from intermediate buffers. Zero-copy communication protocols help improve the performance of DSM systems but because it requires registration of the entire address space, it can be used only for small problem sizes.

- Protected Channel for Communication. The VI architecture requires that a VI be explicitly connected with another VI in order to transfer data between them. Communication using the VI channels established by the connection process eliminates the protection check by the operating system from the critical path of data transfer. This feature is not relevant to software DSM systems that typically assume no sharing of the cluster with other applications.

The VI architecture supports two types of data transfer models for communication. The *Send-Receive* model is similar to traditional message passing, which involves an explicit receive operation, and the recipient of a message has to specify the memory location where the data will be placed. The *Remote Direct Memory Access* (RDMA) model involves only the sender, and no receive operation is required. In this case, both the source and destination buffer are specified by the sender. The VIA specification defines two RDMA operations, RDMA Write and RDMA Read.

3 Software DSM

Software DSM is a runtime system that provides the shared address space abstraction across a message-passing based cluster of computers. The basic idea suggested by Kai Li [21], is to use the virtual memory page protection mechanism to implement an invalidation-based coherence protocol similar to directory-based cache coherence, but at page granularity and completely in software. Since the unit of coherence is a virtual memory page, false sharing occurs when multiple unrelated shared objects lie on the same page. To alleviate the message traffic that would be generated in the presence of false sharing, several relaxed consistency models have been proposed [16, 4, 19, 6, 18]. These consistency models define a memory model for programmers in which they agree to exclusively use explicit synchronization. Under this assumption, the coherence protocol can delay the invalidation messages until a synchronization operation is performed, thus reducing both the protocol messages as well as the extra communication that an early invalidation would have unnecessarily caused.

3.1 Lazy Release Consistency

The most frequently used consistency model in software DSM is Lazy Release Consistency (LRC) [19, 6], in which the invalidations are propagated at acquire time. *Acquire* and *release* are the two explicit synchronization operations required in release consistency model and correspond to lock acquire and lock release respectively. A *barrier* is a global synchronization operation, implemented as a release followed by an acquire. In LRC, the updates are detected in software by computing diffs between the dirty page and a snapshot of the clean copy of the page.

The protocol that we chose to implement on VIA is HLRC [35]. The HLRC protocol implements a multiple-writer scheme by selecting a home for each page, to which updates are sent. The basic idea is to compute diffs at the end of an interval to detect updates and to transfer the updates as diffs to their homes. As a result, the home copy is up-to-date and can be used to update other non-home copies on demand. This protocol has been shown to have very good scalability: the number of messages necessary to update all copies is linear in the number of nodes and the memory overhead is constant [35]. The home-based protocol has also been shown to suit well with user-level memory-mapped communication because pages can be fetched from homes with no copy and diffs can be applied directly on the home's copy [15].

In software DSM, the explicit synchronization operations (acquire, release and barrier) are implemented using message passing. Each lock has a home through which the current owner of the lock is found. Usually, a distributed queue is used to implement queuing for lock acquires. Barriers can be implemented with a linear number of messages using a barrier manager or hierarchically using a logarithmic number of messages. In release consistent software DSM, invalidations are propagated as a list of *write-notices* at synchronization time.

3.2 Basic Programming Model

Typically, software shared memory provides an incomplete shared memory programming model. The execution model is based on multiple threads (one or more on each node) that share static global data in read-only mode, and dynamically allocated data in read-write mode. The coherence applies to the latter exclusively. Static data is usually updated by the main thread before the other threads are spawned. Also, all global shared memory allocations must be performed by the main thread before the other threads are spawned. Since static data cannot be modified once the threads are spawned, it is typically used to maintain pointers to the shared data.

Applications written to use our DSM system make use of the parmacs macros, which were developed at ANL. The macros provide platform independence to the application, enabling it to run on software DSM as well as hardware DSM systems without modification. These macros provide a minimum set of primitives that are necessary in order to program a shared memory application.

The protocol implements the multi-threading model by "forking" one process on each node of the cluster. Each process will execute at least one application thread. The threads will share the address space within the process as well as across the forked processes using software DSM. Since Linux doesn't provide a remote fork, we provide the "illusion" of this by starting the same executable on each node using *rsh*. Each remote process executes the same code as the initial process did before spawning, to initialize static data, making it coherent across nodes.

4 Protocol Design

In this section, we explain the design of the HLRC protocol. We describe the entry points to the protocol by specifying for each entry point the protocol actions and the messages used to perform these actions.

4.1 Protocol Entry Points

Protocol activity occurs at various points in the execution of an application. The entry points to the protocol can be *synchronous* or *asynchronous*. *Synchronous* entry points are those at which the application traps into the protocol and executes some protocol action. The *asynchronous* entry points are entered as a result of incoming messages generated by protocol action on other nodes in the system.

4.1.1 Synchronous Entry Points

During its execution, the application can enter the protocol *synchronously* for the following:

- Lock Acquire - When the application needs a lock, it depends on the underlying HLRC protocol to get the lock from the current owner and perform the appropriate coherence actions.

- Lock Release - When the application needs to release the lock, it uses the HLRC protocol to manage the released lock and perform the appropriate coherence actions.

- Barrier - The application depends on the HLRC protocol to implement a barrier among the participating nodes.

- Page fault - When the application tries to access shared data which has been invalidated as a result of a coherence action, a page fault is generated. The page fault handler, installed by the HLRC protocol at initialization, will fetch the shared page from its home.

4.1.2 Asynchronous Entry Point

The synchronous entry points generate request messages which have to be serviced at the receiving node. The HLRC protocol provides an asynchronous entry point to process the received messages. This can be implemented in several ways:

- Hardware - If support is available in the network interface for asynchronous message handling (for instance with a complete implementation of the VIA specification, a page request can be serviced with an RDMA Read).

- Interrupt Handler - Interrupt handlers can be used to receive and process remote requests if notifications are issued on message arrival.

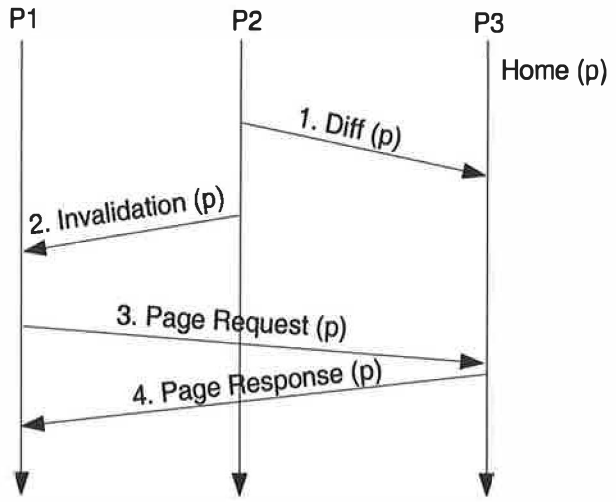


Figure 1: Coherence Messages exchanged by three processes in HLRC

- Communication Thread - A separate communication thread can be used to handle messages, using either polling or blocking.

We use Giganet's implementation of VIA, which does not support RDMA Read or asynchronous notification. Therefore, the asynchronous entry point in our implementation is covered by a separate communication thread on each node that is responsible for handling all the incoming messages.

4.2 Protocol Messages

The protocol activity generates two types of messages: *coherence messages* and *synchronization messages*.

The *coherence messages* are related to update propagation and fall in one of the following categories:

- Diffs - sent by a writer of the page to the home of the page at release or acquire time; contain the updates performed by the sender since the last release or acquire.
- Invalidations - sent at acquire time by the last releaser; contain a list of pages that were updated at the last releaser and elsewhere, that the acquirer must update.
- Page fetch request - sent at page fault time to the home.
- Page response - sent by the home to the faulting node as a response to the page fetch request message.

Figure 1 illustrates the flow of coherence messages when a shared page (p) is updated by a process (P2) and subsequently accessed by another process (P1). The timing and order of the coherence operations are determined by the consistency model implemented by the DSM system. For example, in homeless LRC, the diff messages are sent lazily on demand, while in the home-based LRC, diffs are sent eagerly, either at release or acquire time.

The *synchronization messages* are used to implement the distributed queue for locks and the distributed barrier. In most software DSM protocols, especially in LRC, coherence messages and synchronization messages are combined in a single message whenever possible. For example, in LRC, the invalidation message is combined with the reply message to a lock acquire.

4.3 Implementation of HLRC on VIA

4.3.1 Data Transfers

The data transfers (pages and diffs) are performed using RDMA Write with or without copy. If the problem size is small, the entire shared address space is registered with VIA, and page transfers from home to the non-home nodes are performed without any copy. Memory registration consists of locking the pages of a virtually contiguous memory region into physical memory and providing the virtual to physical translations to the NIC. The amount of physical memory on the machine imposes a limit on the amount of memory that can be registered. If the problem size is larger than the limit imposed by the VIA implementation for memory registration, a set of communication buffers are registered instead and page transfers are performed with one copy at each end (from page to buffer at the sender and from buffer to page at the receiver). Unlike VMMC, another memory-mapped communication library that requires receive buffers to be registered ("exported" in its terminology), VIA requires both send and receive buffers to be registered.

To send an update, the diff is computed by packing all the modified words within a dirty page into one message by the sender (non-home node), and sent to

Applications	Problem Size	Sequential Time (s)	Shared memory size
Barnes-Spatial	262144 bodies	357	325 MB
FFT	2048x2048	86	196 MB
LU	2048 x 2048	209	33 MB
Ocean	514 x 514	30	97 MB
Radix	45M keys	95	377 MB
Water-Nsquared	32768 molecules, 5 steps	22450	22 MB
Water-Spatial	262144 molecules	14202	264 MB

Table 1: Application characteristics

the home of the page. The receiving node applies the diff by modifying the appropriate page at the words mentioned in the diff message.

4.3.2 Remote Requests

The DSM protocol may issue remote requests for data and synchronization. These requests, *which require a response*, are sent using the send-receive model. Since each node executes one application thread, there can be only one outstanding request issued by that node and, one corresponding reply. Therefore, each node expects at most N-1 requests (one from each other node). This means that each node must register N-1 receive buffers and post the same number of receive descriptors, where N is the number of nodes in the cluster. A N-th registered receive buffer is used to receive the reply messages (acks, locks, etc). Since VIA does not support notification on message arrival, a server thread is run on each node, which is responsible to handle remote requests. When no requests are pending, the server thread blocks on a completion queue that aggregates the receive queues for the N-1 buffers on which the node can receive asynchronous requests.

Messages *that do not require a response* (barrier, reply messages) are sent using RDMA Write and do not consume a descriptor on the receiving side. These messages are consumed in a busy loop by the application (not server) thread, since there is nothing else the application thread can do. The memory location for the flag on which spinning is performed, is updated by RDMA Write.

5 Performance Evaluation

5.1 Applications

We evaluated the performance of our DSM system using seven applications from the SPLASH-2 benchmark suite [33]: Barnes, FFT, LU decomposition, Ocean, Radix, Water-Spatial and Water-Nsquared. Due to space limitations, we don't describe the applications in our paper. In Table 1, we show the problem size, sequential execution time and the shared memory footprint for each of these applications.

5.2 Experimental Platform

All our experiments were performed on a cluster of eight SMP PCs. Each PC contains two 300 MHz Pentium II processors. However, for this study, we used only one processor on each node. Each processor has a 512KB L2 cache and each node contains 512 MB of main memory. All nodes run Linux-2.2.10.

Each node has a Giganet cLAN NIC, which is a 32-bit 33 MHz PCI-based card. These nodes are connected by an 8-port Giganet cLAN switch. The performance characteristics for our experimental platform are reported in Table 2. Latency denotes the time taken to transfer a 1 word packet between two nodes using VIA. PostSend denotes the average time taken to post a send using VIA. The last row presents the cost of the VipRegisterMem operation used to register memory used for communication buffers in VIA.

We also present (Table 3) the cost of other operations or events that occur frequently in a software DSM system: page fault handler invocation, the mprotect system call, and memory copy bandwidth.

One-way Latency (1 word)	8.2 μs
Bandwidth (32 KB)	101 MB/s
PostSend (4 KB)	2.1 μs
RegisterMem (4 KB)	4.3 μs

Table 2: Giganet VIA Microbenchmarks

The last row in Table 3 presents the time taken to copy a page(4096 bytes on the Pentium II running Linux) from memory to cache.

Operation (per page)	Time (μs)
Page fault	6.2
Mprotect call	2.7
Memory copy	23.2

Table 3: Linux System Microbenchmarks

In Table 4, we present some microbenchmarks for the DSM system itself. To derive the basic cost of all these operations, these microbenchmarks were done using just two nodes. The Acquire microbenchmark gives the time to update data structures and fetch the lock from a remote node. The Release microbenchmark measures the cost of a release without any pending request for the lock. The page fetch time indicates the time to fetch a page from home without copies. The diff application time includes the time to copy the diff from the diff buffer onto the page and update the version of the page. The Barrier microbenchmark includes the time to send the barrier message to the other node, and wait for the barrier message from the other node.

Operation	Time (μs)
Acquire (Local, Remote)	1, 34
Release	1
Page fetch (no copy)	89
Diff Computation	24
Diff Application	22
Barrier(2-node)	17

Table 4: Software DSM Microbenchmarks

5.3 Application Performance

We ran the seven applications on our cluster of eight nodes. On each node, the application consists of two threads, the communication thread for handling incoming messages and the application thread that performs computation. We present the performance results for the problem sizes mentioned in Table 1 and then analyze the performance in detail.

Table 5 shows the speedups for the seven SPLASH-2 applications we used. LU and Ocean achieved speedups of 7.4 and 7.7 respectively, followed by Water-Spatial, Barnes and Water-Nsquared with speedups greater than 6. FFT comes next followed by Radix which has the worst speedup of the lot.

Applications	Speedup (8 nodes)
Barnes	6.3
FFT	5.8
LU	7.4
Ocean	7.7
Radix	4.3
Water-Nsquared	6.2
Water-Spatial	6.7

Table 5: Speedups on 8 nodes

For the purpose of this study, we classify the applications according to their data access patterns and synchronization behavior. The application can be *single writer* or *multiple writer*, based on the number of concurrent writers on the same coherence unit (a page). The communication to computation ratio is determined by the granularity of data access. *Fine grain* access can introduce fragmentation and/or false sharing, resulting in an increase in the communication to computation ratio. Since all coherence events in the LRC protocols happen at synchronization points, the *frequency of synchronization* plays an important role in the performance. The average computation time between two consecutive synchronization events is a good measure of the frequency of synchronization.

LU and Ocean are single-writer applications with coarse-grain access. These applications exhibit good spatial locality with only one writer per shared page and hence achieve good speedups. FFT is a single-writer application with fine-grained access. The mismatch between the access granularity and the communication granularity prevents it from achieving a better speedup. Applications like Barnes-Spatial and Water-Spatial are multiple-writer with fine-grain access and coarse-grain synchronization. The high average time between synchronization events for these applications helps in achieving good performance. The relaxed consistency model and the multiple-writer support of HLRC helps these applications in achieving good speedups. Water-Nsquared and Radix are multiple-writer applica-

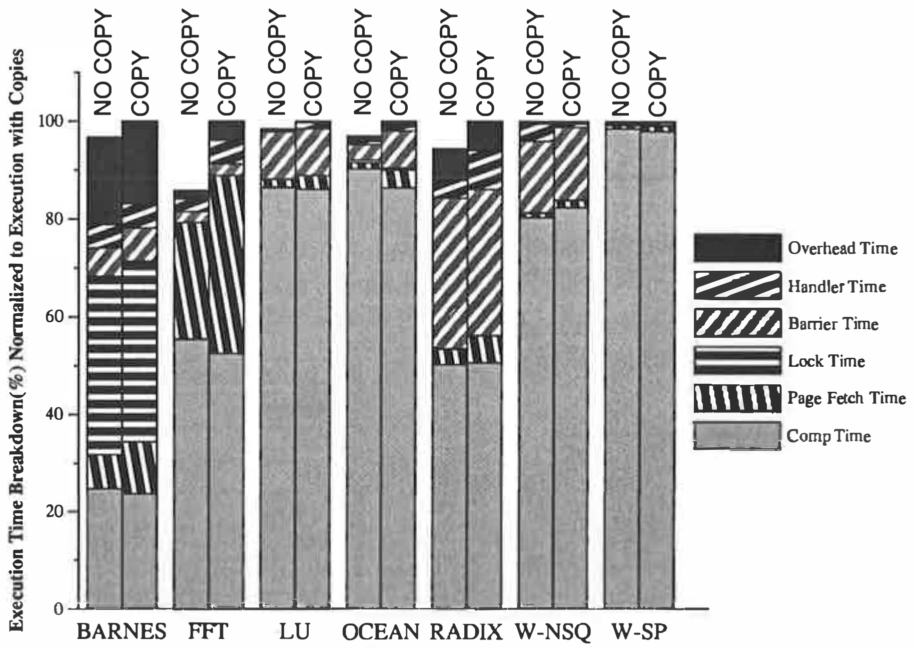


Figure 2: Normalized execution time breakdown on 8 nodes

tions with coarse-grain access. In Water-Nsquared, since each process updates successively a large number of contiguous molecules, the access pattern is preserved at the page level which leads to a coarse-grain access pattern, which is well suited. Radix, however, does not achieve a good speedup due to a large amount of time spent in the barrier, which is caused by an imbalance.

5.4 Performance Impact of Copies

We evaluate the impact of copies, necessary as part of data transfer when the entire shared address space cannot be registered with VIA, by presenting a comparison between the performance of the no-copy and copy versions in Figure 2. We present a comparison of the execution times breakdown for both versions, normalized with respect to the executions with copies. We had to run the applications with problem sizes smaller than the ones mentioned in Table 1 so that we could use both versions with the same problem size. The bars on the left, labeled "NO COPY", present the performance results for the no-copy version, and the bars on the right, labeled "COPY", present the performance results for the version with copies. Each bar presents a percentage breakdown of the different components which make up the execution time on a single node. Computation time is the time spent doing application computation. Page fetch time is the time spent in fetching a page from the home node, on a page miss. Lock time is the time spent in getting the lock from the current owner. Barrier time is the time spent waiting for barrier messages from other nodes, at the barrier. Overhead time is the time spent performing protocol actions. Handler time is the time spent inside the handler, servicing remote requests. Since we used only one processor on each node, for our experiments, the handler competes for the CPU with the application thread to service the messages received via the receive completion queue.

The page fetch time is what increases as a result of the additional copies at the home node and the receiving node during page transfers. We can see that *Page Time* makes up for a significant percentage of the execution time for Barnes, FFT and Radix, and these three applications show an improvement in performance with copy avoidance. Although avoiding copy is good, data transfer with copies doesn't degrade performance drastically. The performance degradation was maximum for FFT (15%) and very little (less than 5%) for the other applications.

6 Discussion

In this section, we present the lessons we learned from this implementation. In particular, we discuss the potential and limitations of the current VIA specification and implementations, for software DSM.

Low-latency Communication. VIA provides low latency communication which is critical for the performance of a DSM system. Figure 3 presents the percentage distribution of the message sizes for four of the applications. For all four applications, small messages (less than 256 bytes in size) constitute more than 75% of the total number of messages.

Copy Avoidance. Copies can be avoided in data transfers but VIA requires both the send and receive buffers to be registered in advance. The cost of memory registration (Table 2) prevents us from doing it at the time of transfer. On the other hand, any VIA implementation imposes a limit on the amount of memory that can be registered. As a result, for large problem sizes, copies cannot be avoided. However, from the results presented in Section 5, we can see that performing copies as part of data transfer doesn't adversely affect application performance except in the case of FFT, where we observed a degradation of roughly 15%.

Scatter-Gather. A scatter-gather mechanism would have been ideal to implement direct diffs without incurring the penalty of multiple message latencies. In the absence of scatter-gather, preliminary calculations indicate that direct diff solutions win over the diff copy solution only when the chunks of consecutive updates are large enough to offset the latency of sending multiple messages using VIA.

To understand the impact of writing diffs directly, avoiding copies but without scatter-gather, we looked at two of the applications, viz., Radix and Barnes which generate a substantial amount of diff traffic. When diffs are written directly, a message is generated for every contiguous dirty segment in the page. Radix achieves an improvement in performance by writing diffs directly, whereas the performance of Barnes degrades. On a careful look at the granularity of the writes and the number of dirty segments per modified page, we realized that Radix resulted in only one contiguous dirty segment per page, whereas Barnes resulted in about 21 dirty segments per page. For Barnes, the overhead of sending multiple dirty segments per page outweighs the improvement achieved by avoiding the copy.

What VIA provides as scatter-gather support is however insufficient for the implementation of direct diffs with one message per page. VIA allows the source of an RDMA Write to be specified as a list of *gather* buffers. However, this gather mechanism doesn't allow us to specify multiple addresses on the destination node. In software DSM, transfer of diffs for any page involves transfer of multiple contiguous dirty segments contained within the page.

We try to estimate the potential performance improvement with scatter-gather support from VIA. We can calculate this by subtracting the time to apply the diff from the handler time. Knowing the total diff size that was transferred and approximating the diff application time with the memory copy time, for all seven applications we studied, we got a gain of no more than 5%. This is consistent with what other people have shown [2].

Remote Read. RDMA Read is a VIA feature that allows fetching of data without interrupting the processor on the remote node. Although present in the VIA specification, the VIA implementation that we used in our experiments does not support RDMA Read. We try to make a rough approximation of the impact of RDMA Read on the performance results.

Using RDMA Read, we can potentially eliminate the handling time for remote requests (since they can be performed by the NIC as an RDMA Read), assuming that RDMA Reads do not require servicing by the CPU. Even though not all remote requests are remote fetches, we look at an upper bound by assuming that the entire handling time is eliminated. For all the applications that we studied, this component (handling incoming messages as a server) of the execution time is not larger than 5%. The elimination of the remote handling time, would also reduce the communication latency experienced by the clients, by the same amount. This brings the total contribution of the remote read to no more than 10%, not counting the side-effect on synchronization due to critical section dilation [2]. Bilas et al [2] have shown that the remote read facility can help reduce the page fetch times by about 20% for most applications.

Broadcast Support. VIA doesn't specify any primitive or mechanism for broadcast. Broadcast can be really useful in the context of a software DSM system. With support for inexpensive broadcast, we can adopt an eager selective update mechanism using broadcast, instead of sending write notices for invalidation. This will help us save unnecessary page requests generated at nodes accessing heavily accessed pages, and in reducing the contention and protocol overhead of serving these pages at the home nodes. We can also broadcast the invalida-

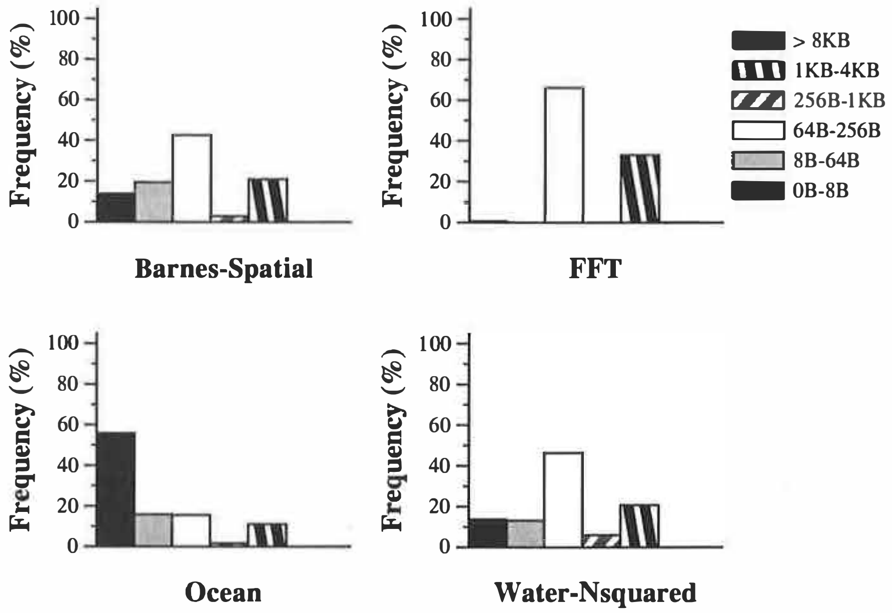


Figure 3: Message size distribution

tions sent at the time of barriers. Previous studies [30] have revealed that a gain of up to 13% could be achieved over 8 nodes, with selective use of broadcast for data used by multiple consumers. They present simulation studies to speculate that a performance improvement of even 50% is possible with 32 nodes.

7 Related Work

This work focuses on using memory-mapped communication to build a high-performance software DSM. In this context, we evaluate VIA as an effective communication substrate for software DSM.

A great deal of work has been done on shared virtual memory since it was first proposed[21]. The Release Consistency (RC) model was proposed in order to improve hardware cache coherence. RC was used to reduce false sharing by allowing multiple writers [4]. Lazy Release Consistency (LRC) [19, 6] further relaxed the RC protocol to reduce protocol overhead. Treadmarks [18] was the first SVM implementation using the LRC protocol on a network of stock computers. The Automatic Update Release Consistency (AURC) [14] protocol was the first proposal to take advantage of memory-mapped communication to implement an LRC protocol. Home-based Lazy Release Consistency (HLRC) [17] proposed a home-based approach to improve the performance on large-scale machines. Cashmere [20] is an eager Release Consistent (RC) SVM protocol that implements a home-based multiple-writer scheme using the I/O remote write operations supported by the DEC Memory Channel network interface [13].

The VI architecture [5] builds on previous work in user-level communication. The VI architecture is based on ideas similar to that of U-Net [11], virtual interfaces to the network from application device channels [7], and Virtual Memory Mapped Communication (VMMC) [8]. Other research that discuss user-level direct access to the network interface are FM [25], AM [10], Hamlyn [32], PM [31], and Trapeze [34].

Prototype implementations of the VI Architecture have been developed on Myrinet, and 100 Mb/s Ethernet. M-VIA [23] is a software emulation of VIA over various network interface cards including Ethernet cards. Berkeley VIA [3] is an implementation of VIA over Myrinet. A performance study of VIA [28] has compared software as well as hardware implementations. The study also explores several performance and implementation issues related to the use of VIA by distributed applications.

Previous work [2, 30, 1] has looked at exploiting support available in hardware to improve the performance of software DSM. Bilas et al [2] explore performance gains to be obtained from performing asynchronous message handling in the network interface. Another study [30] investigates the impact of features such as low-latency messages, protected remote memory writes, inexpensive broadcast and total ordering of network packets on the performance of software DSM. The use of a PCI-based programmable protocol controller for hiding coherence and communication overheads in software DSMs, is studied in [1].

This work sets out to illustrate the match between software DSM requirements and the memory-mapped communication features offered by VIA. To our knowledge, ours is the first performance study of software DSM over VIA.

8 Conclusions

We have implemented a high-performance software distributed shared memory protocol for clusters of PCs connected by Virtual Interface Architecture networks. In this paper, we describe the implementation of a Home-based Lazy Release Consistency DSM protocol on VIA and evaluate its performance on a eight node cluster of PCs using 7 benchmark applications from the Splash-2 suite.

We observe that the VIA primitives are a good match for the requirements of the software DSM communication model. We have learned from our performance study that desirable features for software DSM systems, like scatter-gather, broadcast support, are missing from VIA. Even though the memory registration mechanism imposes a limit on the problem size that can be handled with a zero-copy protocol, our performance studies reveal that copies do not affect the application performance adversely.

The experimental results show that VIA can be successfully used to support shared memory on clusters of PCs but further study is necessary to evaluate its scalability on larger clusters and for a larger set of applications.

Acknowledgments

We thank the referees for their helpful suggestions and our shepherd, Matt Welsh, for working with us to improve the paper. We thank Thu Nguyen, Ricardo Bianchini and Florin Sultan for helpful discussions, and Zoltan Jarai for help in formatting this document. This work was supported by the Rutgers Sciences Council Projects Program, USENIX Research Grant, Microsoft, and NSF CISE-9986046.

Availability

A software distribution package with the software DSM protocol described in this paper is free, and available for download from
`http://discolab.rutgers.edu/projects/dsm`

References

[1] R. Bianchini, L.I. Kontothanassis, R. Pinto, M. De Maria, M. Abud, and C.L. Amorim. Hiding Communication Latency and Coherence Overhead in Software DSMs. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[2] Angelos Bilas, Cheng Liao, and Jaswinder Pal Singh. Using Network Interface Support to Avoid Asynchronous Protocol Processing in Shared Virtual Memory Systems. In *Proceedings of the 26th International Symposium on Computer Architecture*, 1999.

[3] Philip Buonadonna, Andrew Geweke, and David Culler. An Implementation and Analysis of the Virtual Interface Architecture. In *Proceedings of SuperComputing Conference*, November 1998.

[4] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[5] Compaq Corporation, Intel Corporation, and Microsoft Corporation. *Virtual Interface Architecture Specification, Version 1.0. http://www.viarch.org*, 1997.

[6] A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel. Software Versus Hardware Shared-Memory Implementation: A Case Study. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 106–117, April 1994.

[7] Peter Druschel, Bruce S. Davie, and Larry L. Peterson. Experiences with a High-Speed Network Adaptor: A Software Perspective. In *Proceedings of the ACM SIGCOMM Symposium*, September 1994.

[8] C. Dubnicki, L. Iftode, E.W. Felten, and K. Li. Software Support for Virtual Memory-Mapped Communication. In *Proceedings of the 10th International Parallel Processing Symposium*, April 1996.

[9] Cezary Dubnicki, Angelos Bilas, Kai Li, and Jim F. Philbin. Design and Implementation of Virtual Memory-Mapped Communication on Myrinet. In *Proceedings of the 11th International Parallel Processing Symposium*, April 1997.

[10] T. Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, pages 256–266, May 1992.

[11] T. V. Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 40–53, 1995.

[12] GigaNet. *http://www.giganet.com.*

[13] Richard Gillett. Memory Channel Network for PCI. In *Proceedings of Hot Interconnects Symposium*, August 1995.

[14] L. Iftode, M. Blumrich, C. Dubnicki, D.L. Oppenheimer, J.P. Singh, and K. Li. Shared Virtual Memory with Automatic Update Support. In *Proceedings of the 13th ACM International Conference on SuperComputing*, June 1999.

[15] L. Iftode, C. Dubnicki, E. W. Felten, and Kai Li. Improving Release-Consistent Shared Virtual Memory using Automatic Update. In *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[16] L. Iftode and J.P. Singh. Shared Virtual Memory: Progress and Challenges. *Proceedings of the IEEE*, 87(3):498–507, March 1999.

[17] Liviu Iftode. *Home-based Shared Virtual Memory*. PhD thesis, Princeton University, 1998.

[18] P. Keleher, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the Winter USENIX Conference*, pages 115–132, January 1994.

[19] P. Keleher, A.L. Cox, and W. Zwaenepoel. Lazy Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, pages 13–21, May 1992.

[20] L.I. Kontothanassis, G. Hunt, R. Stets, N. Hardavellas, M. Ciernak, S. Parthasarathy, W. Meira Jr., S. Dwarkadas, and M. Scott. VM-based Shared Memory on Low-Latency, Remote-Memory-Access Networks. In *Proceedings of the 24th Annual Symposium on Computer Architecture*, 1997.

[21] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[22] Myricom. *http://www.myri.com.*

[23] National Energy Research Scientific Computing Center. *M-VIA: A High Performance Modular VIA for Linux. http://www.nersc.gov/research/FTG/via*, 1999.

[24] National Energy Research Scientific Computing Center. *MVICH: MPI for Virtual Interface Architecture. http://www.nersc.gov/research/FTG/mvich/index.html*, 1999.

[25] Scott Pakin, Mario Laura, and Andrew Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of SuperComputing Conference*, 1995.

[26] R. Samanta, A. Bilas, L. Iftode, and J.P Singh. Home-based SVM protocols for SMP clusters: Design and Performance. In *Proceedings of the 4th IEEE Symposium on High-Performance Computer Architecture*, 1998.

[27] ServerNet. *http://www.servernet.com.*

[28] W. E. Speight, H. Abdel-Shafi, and J. K. Bennett. Realizing the performance potential of the Virtual Interface Architecture. In *Proceedings of the 13th ACM International Conference on Supercomputing (ICS)*, June 1999.

[29] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L.I. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *Proceedings of the Sixteenth Symposium on Operating Systems Principles*, 1997.

[30] R. Stets, S. Dwarkadas, L.I. Kontothanassis, U. Rencuzogullari, and M. L. Scott. The Effect of Network Total Order and Remote-Write Capability on Network-based Shared Memory Computing. In *Proceedings of the 6th IEEE Symposium on High-Performance Computer Architecture*, 2000.

[31] H. Tezula. Pin-down Cache: A Virtual Memory Management Technique for Zero-copy Communication. In *Proceedings of the International Parallel Processing Symposium*, pages 308–314, 1998.

[32] John Wilkes. Hamlyn – An Interface for Sender-Based Communications. Technical Report HPL-OSR-92-13, Hewlett-Packard Laboratories, November 1993.

[33] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. Methodological Considerations and Characterization of the SPLASH-2 Parallel Application Suite. In *Proceedings of the 23rd Annual Symposium on Computer Architecture*, May 1995.

[34] K. Yocum. Cut-through delivery in Trapeze: An exercise in low-latency messaging. In *Proceedings of the International Symposium on High Performance Distributed Computing*, pages 243–252, 1997.

[35] Y. Zhou, L. Iftode, and K. Li. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. In *Proceedings of the Operating Systems Design and Implementation Symposium*, October 1996.

Performance Comparison of LAM/MPI, MPICH, and MVICH on a Linux Cluster connected by a Gigabit Ethernet Network. *

Hong Ong[‡] and Paul A. Farrell[†]
Department of Mathematics and Computer Science,
Kent State University, Kent, OH 44242.
[‡] hong@mcs.kent.edu [†] farrell@mcs.kent.edu

Abstract

We evaluate and compare the performance of LAM, MPICH, and MVICH on a Linux cluster connected by a Gigabit Ethernet network. Performance statistics are collected using NetPIPE which show the behavior of LAM/MPI and MPICH over a gigabit network. Since LAM and MPICH use the TCP/IP socket interface for communicating messages, it is critical to have high TCP/IP performance. Despite many efforts to improve TCP/IP performance, the performance graphs presented here indicate that the overhead incurred in protocol stack processing is still high. Recent developments such as the VIA-based MPI implementation MVICH can improve communication throughput and thus give the promise of enabling distributed applications to improve performance.

1 Introduction

Due to the increase in network hardware speed and the availability of low cost high performance workstations, cluster computing has become increasingly popular. Many research institutes, universities, and industrial sites around the world have started to purchase/build low cost clusters, such as Linux Beowulf-class clusters, for their parallel processing needs at a fraction of the price of mainframes or supercomputers.

On these cluster systems, parallel processing is usually accomplished through parallel programming libraries such as MPI, PVM [Geist], and BSP [Jonathan]. These environments provide well-defined portable mechanisms for which concurrent applications can be developed easily. In particular, MPI has been widely accepted in the scientific parallel computing area. The use of MPI has broadened over time as well. Two of the most extensively used MPI implementations are MPICH [Gropp, Gropp2] from Mississippi State University and Argonne National Laboratory and LAM [LSC] originally from Ohio Supercomputing Center. LAM is now being maintained by the University of Notre Dame. The modular design taken by MPICH and LAM has allowed research organizations and commercial vendors to port the software to a great variety of multiprocessor and multicomputer platforms and distributed environments.

Naturally, there has been great interest in the performance of LAM and MPICH for enabling high-performance computing in clusters. Large scale distributed applications using MPI (either LAM or MPICH) as communication transport on a cluster of computers impose heavy demands on communication networks. Gigabit Ethernet technology, among others high-speed networks, can in principle provide the required bandwidth to meet these demands. Moreover it also holds the promise of considerable price reductions, possibly even to commodity levels, as Gigabit over copper devices become more available and use increases. However, it has also shifted the communication bottleneck from network media to protocol processing. Since LAM and MPICH use TCP/UDP socket interfaces to communicate messages between nodes, there have been great efforts in reducing the overhead incurred in processing the TCP/IP stacks. However, the efforts have yielded only moderate improvement. Since then, many systems such as U-Net [Welsh], BIP [Geoffray], and Ac-

*This work was supported in part by NSF CDA 9617541, NSF ASC 9720221, by the OBR Investment Fund Ohio Communication and Computing ATM Research Network (OCARnet), and through the Ohio Board of Regents Computer Science Enhancement Initiative

tive Message [Martin] have been proposed to provide low latency and high bandwidth message-passing between clusters of workstations and I/O devices that are connected by a network. More recently, the Virtual Interface Architecture (VIA) [Compaq] has been developed to standardize these ideas. VIA defines mechanisms that will bypass layers of protocol stacks and avoid intermediate copies of data during sending and receiving messages. Elimination of this overhead not only enables significant communication performance increases but will also result in a significant decrease in processor utilization by the communication subsystem. Since the introduction of VIA, there have been several software and hardware implementations of VIA. Berkeley VIA [Buonadonna], Giganet VIA [Speight], M-VIA [MVIA], and FirmVIA [Banikazemi] are among these implementations. This has also led to the recent development of VIA-based MPI communications libraries, noticeably MVICH [MVICH].

The rest of this paper is organized as follows: In Section 2.1, we briefly overview the VIA architecture. In Section 2.2, we give a brief description of Gigabit Ethernet technology. The testing environment is given in Section 3. In Section 4, we present performance results for TCP/IP, LAM and MPICH. Preliminary performance results using VIA and MVICH on a Gigabit Ethernet network will also be presented. Finally, conclusions and future work are presented in Section 5.

2 VIA and Gigabit Ethernet

2.1 VIA Overview

The VIA [INTEL, INTEL2] interface model was developed based on the observation by researchers that the most significant overhead was the time required to communicate between processor, memory, and I/O subsystems that are directly connected to a network. In particular, communication time is not scaling to each individual component of the whole system and can possibly increase exponentially in a cluster of workstations.

This communication overhead is caused by the time accumulated when messages move through different layers of the Internet protocol suite of TCP/UDP/IP in the operating system. In the past, the overhead of end-to-end Internet protocols did not significantly contribute to the poor network performance since the latency equation was primarily dominated by the underlying network links. However, recent improvements in network technology and processor speeds has made the overhead of the Internet protocol stacks the dominant factor in the latency equation.

The VIA specification defines mechanisms to avoid this communication bottleneck by eliminating the intermediate copies of data. This effectively reduces latency and lowers the impact on bandwidth. The mechanisms also enable a process to disable interrupts under heavy workloads and enable interrupts only on wait-for-completion. This indirectly avoids context switch overhead since the mechanisms do not need to switch to the protocol stacks or to another process.

The VIA specification only requires control and setup to go through the OS kernel. Users (also known as VI Consumers) can transfer their data to/from network interfaces directly without operating system intervention via a pair of send and receive work queues. And, a process can own multiple work queues at any given time.

VIA is based on a standard software interface and a hardware interface model. The separation of hardware interface and software interface makes VI highly portable between computing platforms and network interface cards (NICs). The software interface is composed of the VI Provider library (VIPL) and the VI kernel agent. The hardware interface is the VI NIC which is media dependent. By providing a standard software interface to the network, VIA can achieve the network performance needed by communication intensive applications.

VIA supports send/receive and remote direct memory access (RDMA) read/write types of data movements. These operations describe the gather/scatter memory locations to the connected VI. To initiate these operations, a registered descriptor should be placed on the VI work queue. The current revision of the VIA specification defines the semantics of a DMA Read operation but does not require that the network interface support it.

The VI kernel agent provides synchronization by providing the scheduling semantics for blocking calls. As a privileged entity, it controls hardware interrupts from the VI architecture on a global, and per VI basis. The VI kernel agent also supports buffer registration and de-registration. The registration of buffers allows the enforcement of protection across process boundaries via page ownership. Privileged kernel processing is required to perform virtual-to-physical address translation and to wire the associated pages into memory.

2.2 Gigabit Ethernet Technology

Gigabit Ethernet [GEA], also known as the IEEE Standard 802.3z, is the latest Ethernet technology. Like Ethernet, Gigabit Ethernet is a media access control (MAC) and physical-layer (PHY) technology. It offers one gigabit per second (1 Gbps) raw bandwidth which is 10 times faster than fast Ethernet and 100 times the speed of regular Ethernet. In order to achieve 1 Gbps, Gigabit Ethernet uses a modified version of the ANSI X3T11 Fibre Channel standard physical layer (FC-0). To remain backward compatible with existing Ethernet technologies, Gigabit Ethernet uses the same IEEE 802.3 Ethernet frame format, and a compatible full or half duplex carrier sense multiple access/ collision detection (CSMA/CD) scheme scaled to gigabit speeds.

Like its predecessor, Gigabit Ethernet operates in either half-duplex or full-duplex mode. In full-duplex mode, frames travel in both directions simultaneously over two channels on the same connection for an aggregate bandwidth of twice that of half-duplex mode. Full duplex networks are very efficient since data can be sent and received simultaneously. However, full-duplex transmission can be used for point-to-point connections only. Since full-duplex connections cannot be shared, collisions are eliminated. This setup eliminates most of the need for the CSMA/CD access control mechanism because there is no need to determine whether the connection is already being used.

When Gigabit Ethernet operates in full duplex mode, it uses buffers to store incoming and outgoing data frames until the MAC layer has time to pass them higher up the legacy protocol stacks. During heavy traffic transmissions, the buffers may fill up with data faster than the MAC layer can process them. When this occurs, the MAC layer prevents the upper layers from sending until the buffer has room to store more frames; otherwise, frames would be lost due to insufficient buffer space.

In the event that the receive buffers approach their maximum capacity, a high water mark interrupts the MAC control of the receiving node and sends a signal to the sending node instructing it to halt packet transmission for a specified period of time until the buffer can catch up. The sending node stops packet transmission until the time interval is past or until it receives a new packet from the receiving node with a time interval of zero. It then resumes packet transmission. The high water mark ensures that enough buffer capacity remains to give the MAC time to inform the other devices to shut down the flow of data before the buffer capacity overflows. Similarly, there is a low water mark to notify the MAC control when there is enough open capacity in the buffer to restart the flow of incoming data.

Full-duplex transmission can be deployed between ports on two switches, a workstation and a switch port, or between two workstations. Full-duplex connections cannot be used for shared-port connections, such as a repeater or hub port that connects multiple workstations. Gigabit Ethernet is most effective when running in the full-duplex, point-to-point mode where full bandwidth is dedicated between the two end-nodes. Full-duplex operation is ideal for backbones and high-speed server or router links.

For half-duplex operation, Gigabit Ethernet will use the enhanced CSMA/CD access method. With CSMA/CD, a channel can only transmit or receive at one time. A collision results when a frame sent from one end of the network collides with another frame. Timing becomes critical if and when a collision occurs. If a collision occurs during the transmission of a frame, the MAC layer will stop transmitting and retransmit the frame when the transmission medium is clear. If the collision occurs after a packet has been sent, then the packet is lost since the MAC layer has already discarded the frame and started to prepare for the next frame for transmission. In all cases, the rest of the network must wait for the collision to dissipate before any other devices can transmit.

In half-duplex mode, Gigabit Ethernet's performance is degraded. This is because Gigabit Ethernet uses the CSMA/CD protocol which is sensitive to frame length. The standard slot time for Ethernet frames is not long enough to run a 200-meter cable when passing 64-byte frames at gigabit speed. In order to accommodate the timing problems experienced with CSMA/CD when scaling half-duplex

Ethernet to gigabit speed, slot time has been extended to guarantee at least a 512-byte slot time using a technique called *carrier extension*. The frame size is not changed; only the timing is extended.

Half-duplex operation is intended for shared multistation LANs, where two or more end nodes share a single port. Most switches enable users to select half-duplex or full-duplex operation on a port-by-port basis, allowing users to migrate from shared links to point-to-point, full-duplex links when they are ready.

Gigabit Ethernet will eventually operate over a variety of cabling types. Initially, the Gigabit Ethernet specification supports multi-mode and single-mode optical fiber, and short haul copper cabling. Fiber is ideal for connectivity between switches and between a switch and high-speed server because it can be extended to greater length than copper before signal attenuation becomes unacceptable and it is also more reliable than copper. In June 1999, the Gigabit Ethernet standard was extended to incorporate category 5 unshielded twisted-pair (UTP) copper media. The first switches and network NICs using category 5 UTP became available at the end of 1999.

3 Testing Environment

The testing environment for collecting the performance results consists of two Pentium III PCs running at 450MHz with a 100MHz memory bus, and 256MB of PC100 SD-RAM. The PCs are connected back to back via a Gigabit Ethernet NIC installed in the 32bit/33MHz PCI slot. The PCs are also connected together through a SVEC 5 port 10/100 Mbps autosensing/autoswitching hub via 3Com PCI 10/100Mbps Ethernet Cards (3c905B). Three different types of Gigabit Ethernet NICs, the Packet Engine GNIC-II, the Alteon ACEnic, and the SysKonnect SK-NET were tested. The device drivers used are Hamachi v0.07 for GNIC-II, Acenic v0.45 for ACEnic, and Sk98lin v3.01 for SK-NET. In addition, the cluster is isolated from other network traffic to ensure the accuracy of the tests. The cluster is running Red Hat 6.1 Linux distribution with kernel version 2.2.12. In addition, M-VIA v0.01, LAM v6.3, MPICH v1.1.2, and MVICH v0.02 were installed. M-VIA is an implementation of VIA for Linux, which currently supports fast Ethernet cards with DEC Tulip chipset or the Intel i8255x chipset, and the Packet Engines GNIC-I and GNIC-II Gigabit Ethernet cards. The device driver used by M-VIA is a modified version of Donald Becker's Hamachi v0.07. MVICH is a MPICH based MPI implementation for M-VIA. M-VIA and MVICH are being developed as part of the NERSC PC Cluster Project. NetPIPE-2.3 [Snell] was used to test the TCP/IP, LAM, MPICH and MVICH performance. For M-VIA tests, we used the `vnettest.c`, a ping-pong-like C program, distributed with the software.

4 Performance Evaluation

In this section, we present and evaluate TCP/IP, M-VIA, LAM, MPICH, and MVICH point-to-point communication performance using the various Gigabit Ethernet NICs mentioned earlier. The throughput graph is plotted using throughput versus transfer block size. Throughput is reported in megabits per second (Mbps) and block size is reported in bytes since they are the common measurements used among vendors. The throughput graph clearly shows the throughput for each transfer block size and the maximum attainable throughput. The throughput graph combined with application specific requirements will help programmers to decide what block size to use for transmission in order to maximize the achievable bandwidth. The signature (latency) graph is plotted using throughput per second versus total transfer time elapsed in the test. In `NetPIPE`, the latency is determined from the signature graph. The network latency is represented by the time to transfer 1 byte and thus coincides with the time of the first data point on the graph.

4.1 Comparing TCP/IP and M-VIA Performance

Since TCP was originally engineered to provide a general transport protocol, it is not by default optimized for streams of data coming in and out of the system at high transmission rates (e.g 1Gbps). In [Farrell], it is shown that communication performance is affected by a number of factors and indicated that one can tune certain network parameters to achieve high TCP/IP performance especially for a high speed network such as a Gigabit Ethernet network. We have taken care to tune the TCP pa-

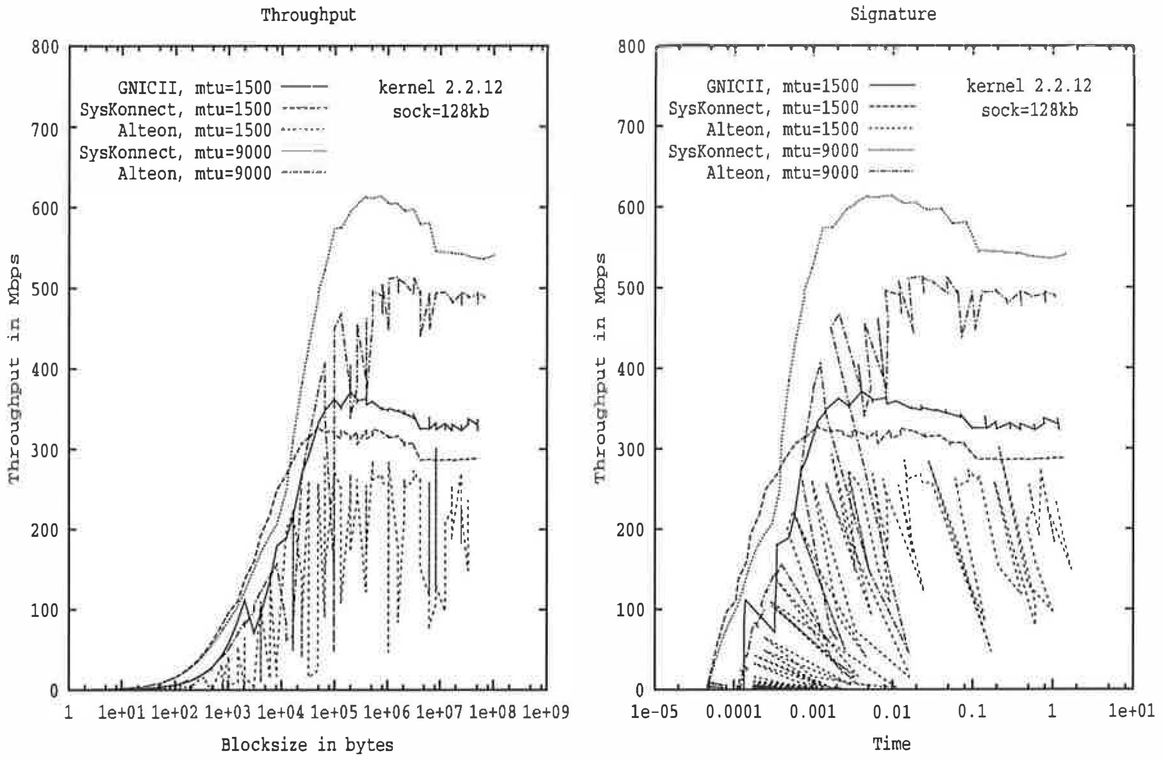


Figure 1: GNIC-II, Alteon, & SysKonnect: TCP Throughput and Latency

rameters according to RFC 1323 TCP/IP Extension for High Performance [RFC1323] in order to achieve high speed TCP/IP communication. We have also set the window size to 128KB rather than the default 64KB in the Linux 2.2.12 kernel.

Figure 1 shows the TCP/IP throughput and latency for various Gigabit Ethernet NICs. Since ACEnic and SK-NET support frame sizes larger than the default of 1500 bytes, we tested them with different MTU sizes. In the figure, we present the TCP/IP performance with MTU of 1500 bytes for all Gigabit Ethernet NICs, and also with MTU equals to 9000 for ACEnic and SK-NET which achieves the highest peak throughput.

One obvious observation from the figures is there are many severe dropouts in ACEnic TCP/IP performance. The reason for these dropouts is supposedly due to the ACEnic device driver. For instance, using ACEnic device driver v0.32, we obtained maximum TCP/IP throughput of 356Mbps using MTU equals to 1500 and 468 Mbps using an MTU of 6500 bytes as opposed to an MTU of 9000 bytes. Furthermore, the latency of ACEnic driver v0.32 is approximately 40% less than the latency of ACEnic device driver v0.45. In addition, with MTU of 1500 bytes and the ACEnic device driver v0.32, the TCP throughput performance is better than that presented here. However, the TCP/IP performance of the ACEnic using device driver v0.45 with large MTU has improved substantially. In general, the overall TCP behavior for both ACEnic device drivers v0.32 and v0.45 have not been improved since v0.28, i.e., the performance graphs have many severe dropouts. In [Farrell], the ACEnic device driver v0.28 running on the Linux 2.2.1 kernel has a smoother performance curve and achieved its maximum throughput of 470 Mbps, using an MTU of 9000.

For MTU of 1500, the maximum attainable throughput is approximately 371 Mbps, 301 Mbps, and 331 Mbps for GNIC-II, ACEnic, and SK-NET respectively. And, the latency is approximately 137 μsecs, 182 μsecs, and 45 μsecs for GNIC-II, ACEnic, and SK-NET respectively. With the lowest latency, SK-NET is able to perform much better than the ACEnic and than the GNIC-II for message sizes up

MTU	Alteon		SysKonnect	
byte	sock=64KB	sock=128KB	sock=64KB	sock=128KB
1500	294.624	301.765	326.488	331.028
4500	402.537	441.932	506.391	509.832
8500	362.068	495.932	495.813	561.456
9000	372.955	513.576	512.490	613.219

Table 1: TCP/IP Performance with Large Socket Buffer and MTU

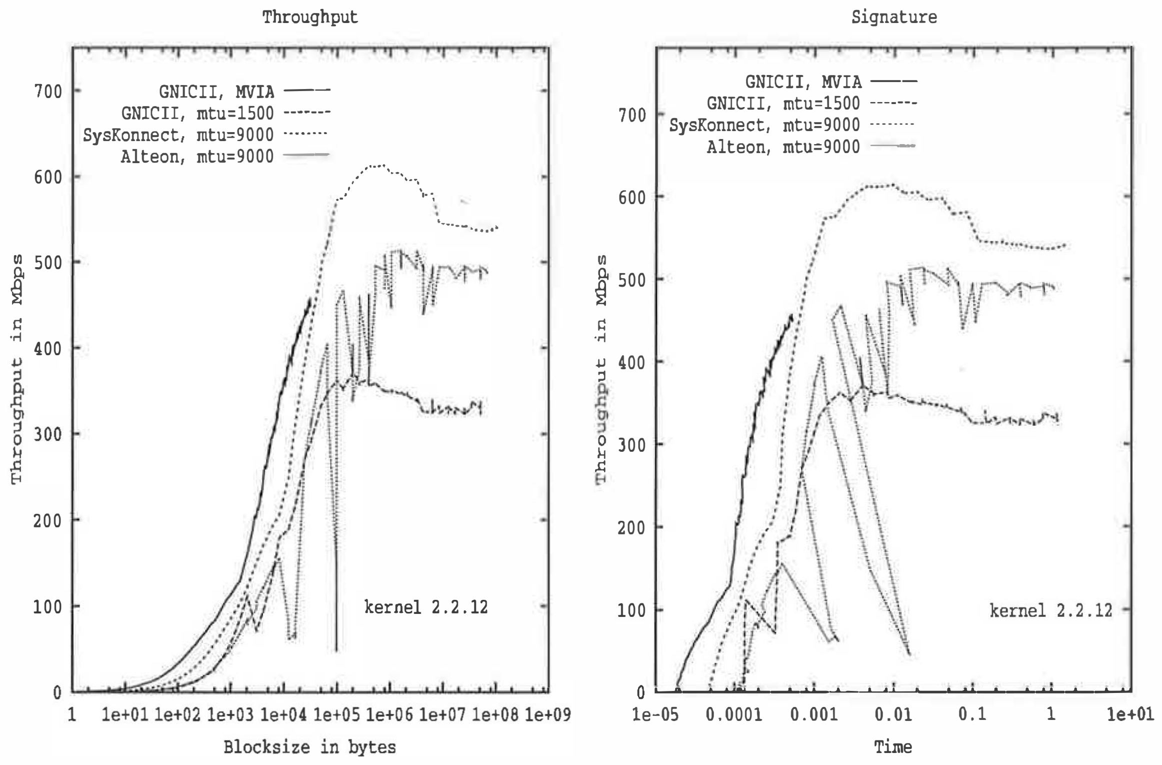


Figure 2: GNIC-II: M-VIA Throughput and Latency

to 49KB. For example, for message size of 16KB, SK-NET throughput is approximately 32% more than the GNIC-II and 82% more than the ACEnic. However, for message sizes greater than 49KB, SK-NET reaches its maximum of 331 Mbps.

Tests on networks based on FDDI, ATM [Farrell2] and Fibre Channel have shown that high speed networks perform better when the MTU is larger than 1500 bytes. Similarly, we expect Gigabit Ethernet would also perform better with an MTU greater than 1500 bytes. From Figure 1, we see that ACEnic maximum attainable throughput increases approximately 70% reaching 513 Mbps when the MTU is set to 9000; And, for SK-NET, the maximum attainable throughput has also increased to approximately 613 Mbps. The latency of ACEnic has decreased to 121 μsecs; and, the SK-NET has increased slightly to 46 μsecs. In order to benefit from the larger MTU, one must also use a larger socket buffer size rather than the default socket buffer size of 64KB. Table 1 shows this effect for various sizes of MTU and socket buffer sizes of 64KB and 128KB.

Figure 2 shows the throughput and latency of M-VIA on the GNIC-II compared with the best attainable performance for each card using TCP. The maximum attainable throughput for M-VIA remains yet to be determined. This is due to the fact that `vnettest.c` stops when message size reaches 32KB which is the maximum data buffer size supported by the M-VIA implementation. For message sizes around 30KB, the throughput reaches approximately 448 Mbps with latency of only 16 μ secs. Thus, the throughput is approximately 53%, 42% and 4% more than GNIC-II, ACEnic, and SK-NET respectively.

The VIA specification only requires VIA developers to support the minimum data buffer of 32KB. However, developers may choose to support data buffer sizes greater than 32KB. In this case, developers must provide a mechanism for the VI consumer to determine the data buffer size. Thus, we expect a larger data buffer will give higher throughput as message sizes continue to increase. On the other hand allocating larger data buffers may result in memory wastage.

4.2 Comparing LAM, MPICH, and MVICH Performance

In this section, we present and compare the performance of LAM, MPICH, and MVICH on a Gigabit Ethernet network. Before moving on to discuss the performance results of LAM and MPICH, it is useful to first briefly describe the data exchange protocol used in these two MPI implementations. The choices taken in implementing the protocol can influence the performance as we will see later in the performance graphs.

Generally, LAM and MPICH use a short/long message protocol for communication. However, the implementation is quite different. In LAM, a short message consisting of a header and the message data is sent to the destination node in one message. And, a long message is segmented into packets with the first packet consisting of a header and possibly some message data sent to the destination node. Then, the sending node waits for an acknowledgment from the receiver node before sending the rest of the data. The receiving node sends the acknowledgment when a matching receive is posted. MPICH (P4 ADI) implements three protocols for data exchange. For short messages, it uses the eager protocol to send message data to the destination node immediately with the possibility of buffering data at the receiving node when the receiving node is not expecting the data. For long messages, two protocols are implemented - the rendezvous protocol and the get protocol. In the rendezvous protocol, data is sent to the destination only when the receiving node requests the data. In the get protocol, data is read directly by the receiver. This choice requires a method to directly transfer data from one process's memory to another such as exists on parallel machines.

All the LAM tests are conducted using the LAM client to client (C2C) protocol which bypasses the LAM daemon. In LAM and MPICH, the maximum length of a short message can be configured at compile time by setting the appropriate constant. We configured the LAM short/long messages switchover point to 128KB instead of the default 64KB. For MPICH, we used all the default settings. Figure 3 shows LAM throughput and latency graphs. And, Figure 4 shows MPICH throughput and latency graphs.

For LAM using MTU size of 1500 bytes, the maximum attainable throughput is about 216 Mbps, 188 Mbps, and 210 Mbps with latency of 140 μsecs, 194 μsecs, and 66 μsecs for the GNIC-II, ACEnic, and SK-NET respectively. For MPICH using MTU size of 1500, the maximum attainable throughput is about 188 Mbps, 176 Mbps, and 249 Mbps with latency of 142 μsecs, 239 μsecs and 99 μsecs for the GNIC-II, ACEnic, and SK-NET respectively. Since LAM and MPICH are layered above TCP/IP stacks, one would expect only a small decrease in performance. However, the amount of performance degradation in LAM and MPICH as compared to the TCP/IP performance is considerable. For LAM, the performance drop of approximately 42%, 38% and 41% for GNIC-II, ACEnic, and SK-NET respectively. And, the performance drop for MPICH is approximately 49%, 42%, and 25% for GNIC-II, ACEnic, and SK-NET respectively.

Changing MTU to a larger size improves LAM performance somewhat. For LAM, the maximum attainable throughput is increased by approximately 42% for SK-NET and by approximately 36% for the ACEnic with MTU of 9000 respectively. However, changing MTU to a bigger size decreases MPICH performance. For MPICH, the maximum attainable throughput drops by approximately 7% for an SK-NET and by approximately 15% for an ACEnic with MTU of 9000.

In all cases, increasing the size of the MTU also increases the latency slightly except in the case of the test on MPICH using the ACEnic. In particular, the latency of LAM is approximately 69 μsecs for SK-NET and 141 μsecs for ACEnic. And, the latency of MPICH is approximately 100 μsecs for SK-NET and 2330 μsecs for ACEnic.

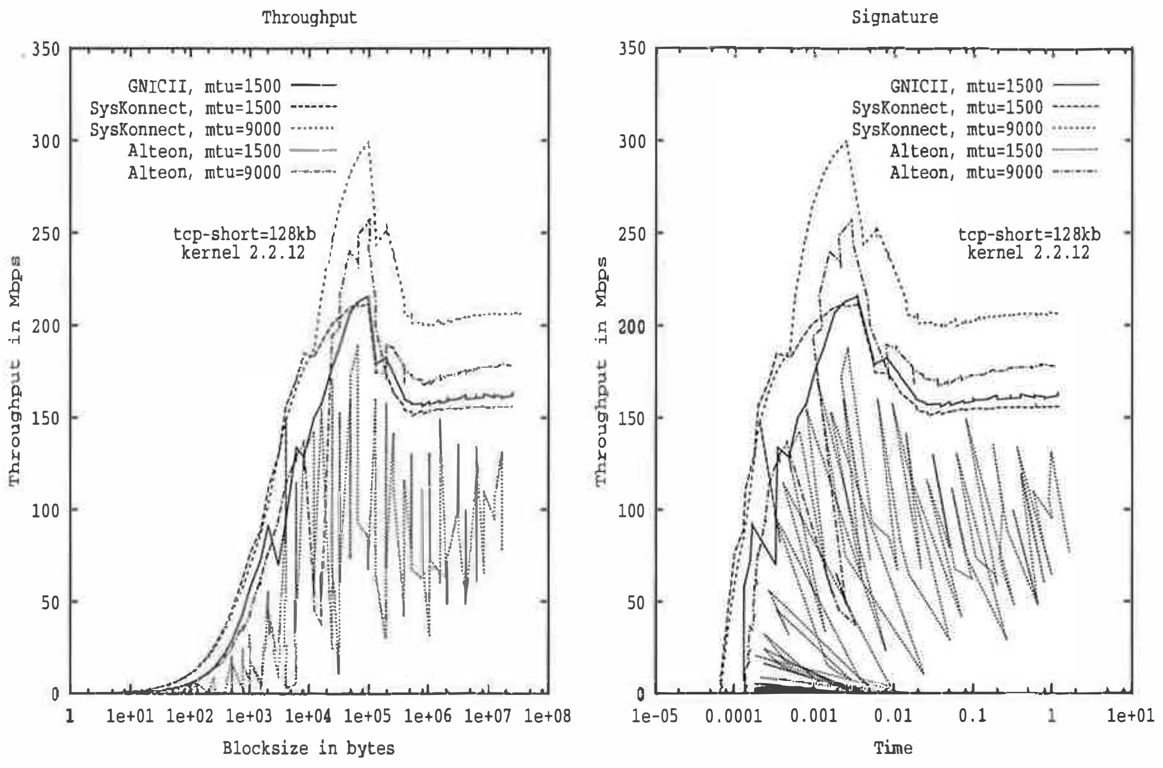


Figure 3: GNIC-II, Alteon, & SysKonnect: LAM Throughput and Latency

Again, we see that there are many severe dropouts for both LAM and MPICH using the ACEnic card.

Several things can be said regarding these performance results.

- As noted from Table 1, TCP/IP performs better on a Gigabit Ethernet network for large MTU and socket buffer size. During initialization, LAM sets send and receive socket buffers, SOCK_SNDBUF and SOCK_RCVBUF, to a size equal to the switch-over point plus the size of the C2C envelope data structure. This explains why, when we made the MTU greater than 1500 bytes, LAM performance improved. However, MPICH initializes SOCK_SNDBUF and SOCK_RCVBUF size equal to 4096 bytes. Hence, a larger MTU does not help to improve MPICH performance much.

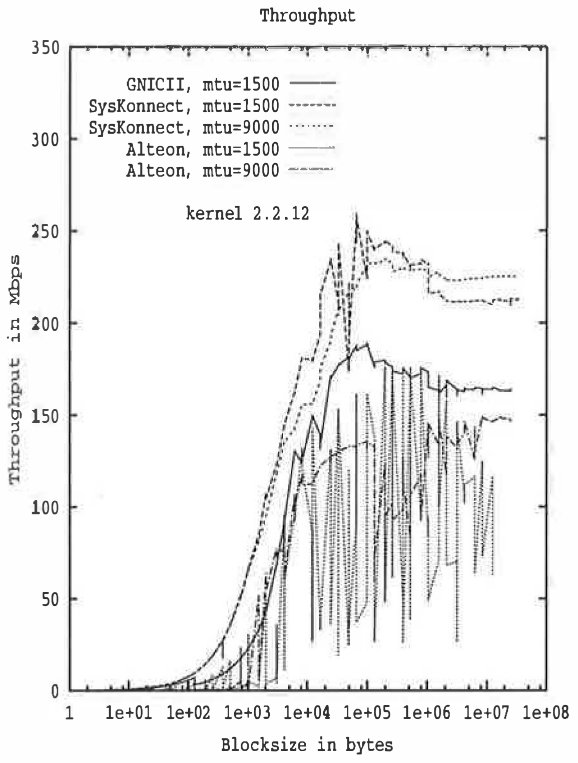
- In both LAM and MPICH, a drop in performance, more noticeably for LAM at 128KB, is caused by the switch from the short to long message protocol described above. In particular, we specified that messages of 128KB or longer be treated as long messages in LAM. For MPICH, the default switch-over point to long message handling is 128000 bytes.

From the figures, it is evident that an MPI implementation layered on top of a TCP/IP protocol depends highly on the underlying TCP/IP performance.

Figure 5 shows MVICH performance. MVICH attains a maximum throughput of 280 Mbps with latency of only 26 μsecs for message sizes as low as 32KB. Again, we were unable to run message sizes greater than 32KB. From the figure, it is evident that, as hoped, MVICH performance is much superior to that of LAM or MPICH using TCP/UDP as communication transport.

5 Conclusion

In this paper, we have given an overview of VIA and Gigabit Ethernet technology. The performance of TCP, M-VIA, LAM, MPICH, and MVICH using three types of Gigabit Ethernet NICs, the GNIC-



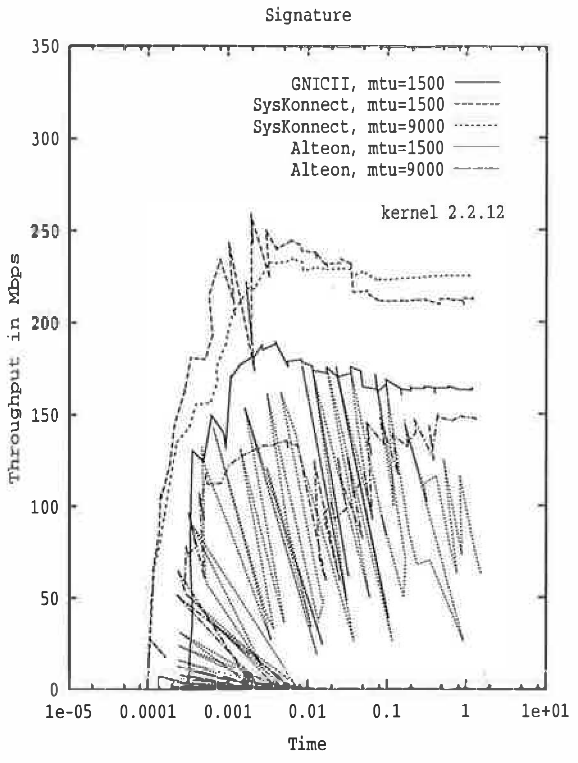


Figure 4: GNIC-II, Alteon & SysKonnect: MPICH Throughput and Latency

II, the ACEnic, and the SK-NET on a PC cluster were presented. We evaluated and compared the performance of TCP, LAM, MPICH, and MVICH. In order to achieve high TCP/IP performance on a high speed network, we indicated that one has to tune certain network parameters such as RFC1323, socket buffer size, and MTU. We attempted to explain the poor performance of LAM and MPICH and show MVICH as a promising communication library for MPI based applications running on a high speed network. We remark that we tested M-VIA v0.01 and MVICH v0.02 which are very early implementations and the performance is likely to improve further with further development.

Further investigation of the effects of tuning the MPICH implementation is also warranted. In addition, ANL is currently in the process of modifying MPICH to provide a device independent layer which will permit easy addition of device driver modules for various networks. This may also lead to further improvements in performance. SysKonnect is currently writing a VIA driver for the SK-NET card. It will be of some interest to compare the throughput and, in particular, latency achievable with this implementation.

References

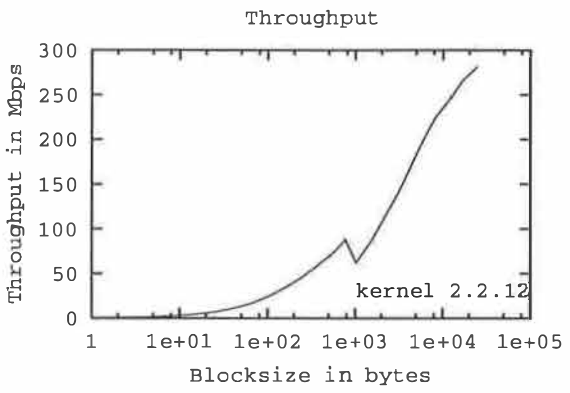
[Banikazemi] M. Banikazemi, V. Moorthy, L. Hereger, D. K. panda, and B. Abali. *Efficient Virtual Interface Architecture Support for IBM SP Switch-Connected NT Clusters.* International Parallel and Distributed Processing Symposium. (2000)

[Buonadonna] P. Buonadonna, J. Coates, S. Low, and D. E. Culler., *Millennium Sort: A Cluster-Based Application for Windows NT using DCOM, River Primitives and the Virtual Interface Architecture.* In Proc. of 3rd USENIX Windows NT Symposium. (1999)

[Compaq] Compaq Computer Corp., Intel Corporation, Microsoft Corporation, *Virtual Interface Architecture Specification version 1.0.* `http://www.viarch.org`

[Farrell] Paul Farrell and Hong Ong, *Communication Performance over a Gigabit Ethernet Network*, IEEE Proc. of 19th IPCCC (2000).

[Farrell2] Paul Farrell, Hong Ong, and Arden Ruttan, *Modeling liquid crystal structures using*



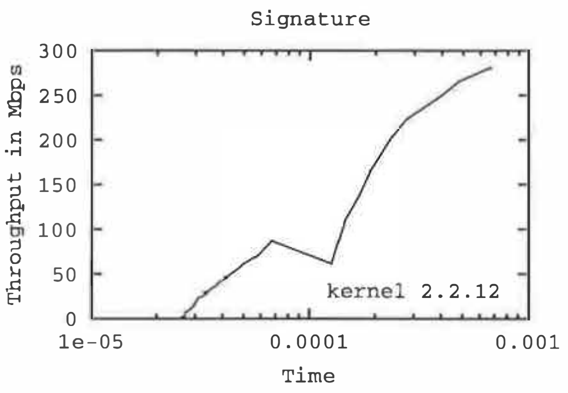


Figure 5: GNIC-II: MVICH Throughput and Latency

MPI on a workstation cluster, to appear in Proc. of MWPP 1999.

[GEA] Gigabit Ethernet Alliance, *Gigabit Ethernet Overview.* (1997)
`http://www.gigabit-ethernet.org/`

[Geist] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R .Manchek, V. Sunderam. *PVM: Parallel Virtual Machine - A User's Guide and Tutorial for Network Parallel Computing.* MIT Press (1994).

[Geoffray] P. Geoffray, L. Lefevre, C. D. Pham, L. Prylli, O. Reymann, B. Tourancheau, and R. Westrelin. *High-speed LANs: New environments for parallel and distributed applications.* In EuroPar'99, France. Springer-Verlag (1999).

[Gropp] W. Gropp and E. Lusk and N. Doss and A. Skjellum, *A high-performance, portable implementation of the MPI message passing interface standard*, Parall. Comp. 22 (1996).

[Gropp2] William D. Gropp and Ewing Lusk, *User's Guide for* `mpich`, *a Portable Implementation of MPI*, Argonne National Laboratory (1996), ANL-96/6.

[Jonathan] Jonathan M. D. Hill, Stephen R. Donaldson, and David B. Skillicorn, *Portability of performance with the BSPlib communications library*, Massively Parallel Programming Models Workshop (1997).

[INTEL] Intel Corporation, *Virtual Interface (VI) Architecture: Defining the Path to Low Cost High Performance Scalable Clusters.* (1997)

[INTEL2] Intel Corporation, *Intel Virtual Interface (VI) Architecture Developer's Guide revision 1.0.* (1998).

[LSC] Laboratory for Scientific Computing at the University of Notre Dame.
`http://www.mpi.nd.edu/lam`

[Martin] Richard P. Martin, Amin M. Vahdat, David E. Culler, Thomas E. Anderson. *Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture.* ISCA 24 (1997).

[MVIA] M-VIA: A High Performance Modular VIA for Linux.
`http://`
`www.nersc.gov/research/FTG/via/`

[MVICH] MPI for Virtual Interface Architecture.
`http://`
`www.nersc.gov/research/FTG/mvich/`

[RFC1323] Jacobson, Braden, & Borman, *TCP Extensions for High Performance*, RFC 1323 (1992).

[Speight] E. Speight, H. Abdel-Shafi, and J. K. Bennett. *Realizing the Performance Potential of the Virtual Interface Architecture.* In Proc. of the International Conf. on Supercomputing. (1999)

[Snell] Q. O. Snell, A. R. Mikler, and J. L Gustafson, *NetPIPE: Network Protocol Independent Performance Evaluator.*, Ames Laboratory/ Scalable Computing Lab, Iowa State. (1997)

[Welsh] Matt Welsh, Anindya Basu, and Thorsten von Eicken. *Incorporating Memory Management into User-Level Network Interfaces.* Proc. of Hot Interconnects V, Stanford (1997).

Large Scale Linux Configuration with LCFG

Paul Anderson
Division of Informatics,
University of Edinburgh
paul@dcs.ed.ac.uk

Alastair Scobie
Division of Informatics,
University of Edinburgh
ajs@dcs.ed.ac.uk

This paper describes the automatic installation and configuration system currently being used to manage several hundred Linux machines in the Division of Informatics at Edinburgh University. This is a development of the LCFG system which has been used successfully for several years under Solaris. The introduction provides some background on the general problem of large-scale configuration, together with a short comparison of typical solutions, and a brief description of the original LCFG system.

The specific changes required to support Linux are then discussed; in particular, the issues of installation bootstrapping, and the *updaterpms* program. This automatically synchronises client software packages with a specification in the central database. We describe how the system is used in practice, and how it enables us to automatically maintain large numbers of machines with very diverse and evolving configurations.

Some future plans are then discussed, including a major reworking of the LCFG implementation, LDAP integration, and our intention to make the technology more widely available.

1 Background

For many years, the Unix community has recognised the inadequacy of vendor-supplied configuration tools for managing large networks of disparate machines. A wide range of solutions have been proposed and developed by systems administrators, frequently just for their own use. These range from simple cloning mechanisms to highly flexible systems (a survey of some previous techniques is available in [1]).

The LCFG framework [3] was developed by The Department of Computer Science at Edinburgh University to handle their own network of several hundred (mostly Solaris) Unix machines. This system was designed to satisfy several fundamental properties that we could not find in any existing implementation, and has been very successful. Over the last three years, the site has migrated rapidly towards Linux, and the LCFG framework has been ported and extended to support this with good results. The Department has recently merged with several others to form the Division of Informatics and work is underway to extend the use of LCFG to the larger domain. As part of this work, we are taking the opportunity to improve the design and implementation in some areas.

2 System Configuration

When a standard release of a large software product, such as an operating system, is distributed to many users, it invariably needs *configuring* to tailor it to the requirements of each individual installation. Even within a single site, there can be a huge difference between configurations of different machines; the following are some examples of the parameters which may vary:

- Hardware configuration and drivers.
- Network configuration and servers.
- Installed software.
- Network services provided.
- Access control.

2.1 Manual Configuration

Obviously the amount of variety between machines depends on the type of installation; an academic Computer Science environment tends to have a larger variety of software and configurations than a site concerned primarily with a single application. Our current LCFG system supports over 2000 parameters, about 25% of which routinely vary between different systems. For individual machines, or a small site, these parameters would traditionally be configured manually, and most distributions include graphical tools to make this process more straightforward (for example [9, 17]). However, large installations require more sophisticated techniques:

2.2 Automatic Configuration

Most obviously, the effort required to configure several hundred machines manually from a graphical interface is not normally acceptable. However, most sites will also want to have a reasonable confidence in the correctness of their configurations; misconfigured systems represent serious security problems, as well as leading to unpredictable failures. Manually configured systems, with no explicit representation of the configuration, are notoriously difficult to guarantee correct. Early attempts to overcome these problems were often based on a *cloning* procedure where a single machine is configured by hand and the resulting disk image is copied directly onto a set of other machines. This is usually followed by execution of some scripts to apply any machine-specific differences. This process is useful for large numbers of very similar machines which do not change regularly, such as those in a student laboratory. It is also widely used in Windows environments.

In many installations, such as our own, both the variety of different configurations, and the rate at which they change, makes cloning impractical. We support a range of machines from file servers, to student laboratory clients, to researcher's laptops, and the hardware and software requirements are all very different. New machines arrive continuously, old machines are reallocated, and systems are rebuilt after hardware failures or OS upgrades; all of these imply a reconfiguration, and we estimate that, on average, about 10% of our machines are completely reconfigured each week. Small configuration changes also occur very frequently in a complex environment; for example, changing a server or gateway can imply configuration changes for many other hosts. Software updates also occur at an average rate of several tens of packages per day.

2.3 Supporting Diversity & Change

Automatically supporting such diversity and rate of change requires two main features from a configuration system: Firstly, there must be some representation of the configuration information which is stored independently of the host systems. This may be simple copies of machine configuration files stored on a central server, or it may be a more complex "database". Secondly, the system must be capable of tracking changes to the configuration and applying them to individual machines as necessary, rather than requiring an explicit reconfiguration operation. *cfengine* [7, 6] is a popular system which addresses this problem.

Both the format and the structure of the configuration information are very important. An obvious solution is to store configuration files in the same format as they appear on the target host, but this coarse grain approach means that the configuration system is not aware of the relationships between data in the various files; for example, the "owner" of a machine may have special access rights which involves the username appearing in several different configuration files, and we would like to be able to change this at a single point. Storing the configuration specifications in a more abstract format allows the configuration to be examined and manipulated in a more meaningful way. It also provides a degree of platform independence, analogous to using a high-level programming language which can be transformed into code for any specific platform. The configuration system may use this information to generate traditional configuration files or the services on the host machine may be modified to read this configuration format directly.

An object-oriented structure is usually the most convenient way of organising the configuration information. Hosts usually fall into various different categories (laptop, web server, student desktop, etc.) and it is natural to specify new machines by using inheritance to describe just the difference (if any) between the new machine and some existing category. Many systems adopt this approach with varying degrees of sophistication (for example [15, 14, 16]). Once the information is available in this high-level form, there is considerable potential for analysing and generating the specification automatically. This opens up the possibility of treating the configuration of a whole site as a complete entity; for example, we should be able to prevent a gateway being removed while there are still clients which depend on it.

The process of actually changing a machine configuration to match a particular specification is not normally straightforward. Ideally, we would like this to take place automatically as soon as the specification is changed. Sometimes this is possible; for example changing the an entry in a TCP wrapper. However, changing the disk partitioning is probably not desirable, or even possible, while a machine is in use. In practice, changes take place at different times, as appropriate; sometimes immediately, sometimes from a nightly cron job and sometimes at reboot. Laptops are an interesting case because they can normally only be reconfigured while they are connected to the network, and this might not happen very often at boot time, or at the time when a nightly cron job would normally run. We allow laptops to be reconfigured on demand by the user so that updates take place at a convenient time.

3 LCFG

LCFG was designed to handle automated installation and configuration in a very diverse and evolving environment. Abstract configuration parameters are stored in a central repository where they are organised in files based on machine categories. A simple inclusion mechanism provides inheritance, as well as a form of modified inheritance which we call *mutation*. The centralised configuration repository and the abstract representation of configuration parameters are key features of LCFG.

A collection of scripts on the host machine read these configuration parameters and either generate traditional configuration files, or directly manipulate various services.

3.1 Configuration Parameters

The configuration parameters are stored in the form of key-value pairs, inspired by X resources, and similar to the parameters used by COAS. For example:

```
mambo.dns.servers localhost
```

The key specifies the hostname, the *subsystem* and the parameter. The configuration files are passed through the C preprocessor, supporting simple inheritance by file inclusion:

```
#include <standard_laptop.h>
auth.owner paul
....
```

The machine-specific file need only list those resources which are different from a standard laptop (the first component of the resource keys is generated automatically from the name of the file). Notice that the included *class* file relates to a high-level concept (standard laptop machine) which may contain resource specifications for many different low-level subsystems. The class files may of course be nested.

Together with the use of preprocessor variables, this simple mechanism provides a powerful way of presenting complex host configurations in a clear way, with very little specialised software. However, it is not sufficiently fine-grained to process information inside the resource keys. This causes difficulties in some cases; for example, a standard configuration might specify that the CD-ROM should have its ownership changed to match that of the user at the console:

```
auth.consolepermclass_cdrom
  /dev/cdrom
```

If we have a second SCSI CD on our machine then we might want to specify:

```
auth.consolepermclass_cdrom
  /dev/cdrom /dev/scd0
```

Specifying this directly for a particular host is not good, because it overrides the specification for the standard machine, so that changes to the standard specification (such as changing the location of the default CD-ROM) would not be reflected on this particular host.

At the same time as porting LCFG to Linux, we added the facility to specify a regular expression for transforming any inherited resource value. This allows us to easily append or prepend items to a standard value:

```
auth.consolepermclass_cdrom
  !/(.*)/\$1 /dev/scd0
```

We call this process *mutation*. Although it is very powerful, overuse can lead to configurations which are very hard to understand, and we usually restrict its use to a few well-defined macros:

```
auth.consolepermclass_cdrom
  ANDALSO(/dev/scd0)
```

The files containing the configuration parameters for the entire installation are maintained on a central server under RCS control. When changes are made, the parameters are preprocessed into a single table which is distributed to the clients as an NIS map. This provides a replicated database which is easily accessible to all machines and simple to implement, but it was only originally intended as a temporary solution and it has a number of problems. These problems and a possible replacement technology are discussed later (section 6). Since this configuration information must be available at boot time, laptops are all configured as NIS slaves which update their maps on demand.

3.2 Subsystem Scripts

Each *subsystem* on a host has a controlling script which is very similar to the startup scripts used by the System V init mechanism. These scripts accept a number of *methods* such as `start` and `stop` which are invoked at appropriate times. Each script reads configuration parameters from the repository and configures the appropriate subsystem. This may involve translating the configuration parameters into a traditional configuration file, or controlling a service directly; for example, starting some daemon with command-line parameters derived from the configuration resources.

Traditionally, these scripts have been simple shell scripts although, under Linux, Perl is available at install time and many scripts now include some Perl code as well. A set of default routines are available which can be included into the script to load resources and perform other utility functions, such as retrieving default values for missing resources. Most scripts are quite short and it is easy to write a configuration script for a new package or service. Initially, new scripts are normally written just to support the subset of configuration parameters which are expected to vary in our installation, and this is easily expanded later as the need arises. The independence, and ease with which these scripts can be created has been a major reason for the success of the system; they are usually created by the person responsible for the corresponding service, and many people have contributed.

The example in section 8 shows a section of code from a script which starts the Samba server for VMware. Notice that most of the numerous Samba parameters are hardwired into a template configuration file, but those which we expect to vary between machines are set from the LCFG resources by a simple substitution of variables in the template. Some of these are also generated from a "higher-level" LCFG parameter which specifies whether or not we want the server to be visible on the external network. Notice also, that VMware itself is started using the standard init script.

3.3 Script Execution

As mentioned earlier, it is not always obvious when a particular service should be reconfigured. Should a daemon be stopped and restarted if necessary to force a reconfiguration? Or should we restart it during the night? Or should we wait until the next reboot? In practice, we never force a reconfiguration immediately whenever a specification changes (although this would not be hard to do). Those services which can be reconfigured while a machine is running, are normally reconfigured nightly by a cron job. Other services are reconfigured at boot time. It is possible for configuration changes to schedule a night-time reboot if it is essential that they are implemented as soon as possible.

A simple client-server application (`om`/`omd`) allows scripts on any remote machine (or group of machines) to be executed manually.

3.3.1 The boot Script

A subsystem script called `boot` is invoked both from the system init scripts, and from a regular `cron` job. This inspects LCFG resources to determine which other subsystems should be run at each stage. The `start` and `stop` methods for each specified script are called by `boot` at changes in the runlevel as specified by the appropriate resource. The `run` method is called at specified intervals, using `cron`. The resources can therefore determine which subsystems are reconfigured, and when.

3.3.2 The update Script

The `update` script controls the updating of software packages. This is described in more detail below and completely replaces the `updatelf` script previously used under Solaris to update software using `lfu` [2]. `update` is also capable of running at install time when it uses LCFG resources to configure those aspects of the machine which are too difficult to change while the system is running; for example, the primary network address and the disk partitioning. The installation process is described in more detail below (section 4.1).

4 Linux-Specific Issues

A number of small improvements were made to various aspects of LCFG when porting to Linux, and two areas were changed significantly:

- The installation bootstrapping process has been changed to accommodate the differences in hardware, and to make use of the new updating mechanism:
- Updating of software uses a completely different technique to takes advantage of the Redhat Package Manager (RPM) [4] and the wide availability of packages in this format.

4.1 Installation Bootstrapping

Providing the ability to install new machines with the absolute minimum of manual intervention is very important. This allows failed machines to be replaced, and new machines to be installed, quickly and correctly, by unskilled staff [1].

Installing an operating system on to bare hardware requires some sort of bootstrap process. Typically:

- A minimal version of the operating system or other install program is loaded from the network, or removable media.
- Once booted, this program partitions the system disk and installs a copy of the operating system.
- There will usually be some additional software installation and configuration, the first time that the machine reboots from the newly installed system.

The first operation tends to be operating system-dependent and the original LCFG made use of Solaris Jumpstart [15, 14] to boot an initial image from the network. The current Linux port requires a boot floppy (or CD) for the same purpose, since not all hardware supports network booting (Kickstart [12] is not used).

Desktop machines may not have CD drives, and they use BOOTP/DHCP to mount the root filesystem from NFS. Laptops cannot use NFS so easily in this way because of the need for PCMCIA drivers, and they are installed using a bootable CD which contains an equivalent image. Once the system is booted, access to the network is necessary to retrieve the configuration parameters from the NIS and the RPMs for building the system disk.

When the minimal system has booted, an `update` script runs automatically to partition the system disk according to the LCFG resources and load the software. In the original Solaris implementation, a small hand-crafted image was first copied directly from the network onto the system disk, but this is difficult to maintain. Under Linux, the `update` script builds the root filesystem completely from a set of RPMs. Although it is aware of the context, this uses exactly the same process which is used nightly to update the software on a running machine (see 4.2.1), ensuring a consistent interpretation of the LCFG resources at install and update time.

When the software has been installed, the system reboots from the new image. The LCFG subsystem scripts start normally and perform the remaining configuration.

Once the install operation has been started, it runs completely unattended, allowing whole laboratories of machines to be installed easily by one person. However, the server load imposed by large numbers of clients performing simultaneous installations can be a performance problem. A "helper" CD is sometimes used to provide a local copy of many of the packages. If the CD becomes out of date, and newer versions of some packages are available on the network, the newer versions will automatically be installed instead. It would be interesting to look at ways of improving simultaneous large-scale installations, perhaps by using multicast to distribute the RPMs.

4.2 Software Updating

The original version of LCFG used a program called `lfu` [2] to update software on the local disks. At that time, clients tended to have smaller disks and mount much of their software from the network, so `lfu` was mainly concerned with synchronising a comparatively small number of replicated servers, and the performance would be poor for a large number of clients. There was also no obvious candidate for a package format which was widely supported by the packages that we wanted to install; `lfu` provides a crude mechanism for matching files to their packages, based on file ownership, but this was not always adequate. More importantly, it was also difficult to keep track of the link between binaries and their corresponding sources.

In theory, `lfu` could have been used under Linux, but the Redhat distribution is based on the RPM package management software which provides a much better mechanism for managing software packages. Many of the packages that we require are also distributed in this format.

Although there are now numerous programs available for updating and distributing RPMS (for example [13, 11, 5, 10]), few of these tools were available in 1997. We also wanted to interface with the LCFG and provide automatic installation, upgrade and deletion of RPMs. The Linux version of LCFG therefore uses a locally-developed program called `updaterpms`. This is currently written in C and makes use of `rpmlib` [2] (rewriting this in Python is a possibility now that there an `rpmlib` interface available).

4.2.1 updaterpms

`Updaterpms` compares the RPMs installed on a machine with a specification provided by the LCFG and installs, upgrades, or deletes RPMs as necessary. Using the current NIS implementation of the LCFG resource map, it would be unwieldy to hold the full list of RPMs in the

[1]This does not include the restoration of any user data from backup

[2]The latest version of `updaterpms` needs a modified version of `rpmlib` to support per-package options

map itself, so the lists are held in files which are referenced by LCFG resources. This is adequate since we tend to install most available software on most machines, but it is not ideal, and this may change as we move to a different technology for map distribution.

The package specification files are preprocessed with the C preprocessor to provide some degree of structure similar to the LCFG files themselves. Individual machines can therefore include a standard software specification and override individual packages. The RPM specifications may contain wildcards to refer to the latest version (or release); for example, the standard installation might include a specific version:

```
toshutils-1-1.34
```

And a particular machine might override that to carry the latest available version:

```
#include <standard>
+toshutils-1-*
```

The '+' symbol indicates that the new specification overrides any preceeding one thus inhibiting the error message that would normally be generated by the duplicate package specifications.

The ability to import software which has been prepackaged in RPM format saves a considerable amount of work. However the packaging is not always well implemented; post-install scripts, for example, are often poorly designed, perhaps attempting to add users to a password file, or to demand user interaction, neither of which are appropriate for an automated install on a networked system. Occasionally, dependency information is also incorrect. Several standard RPM options such as `--noscripts` or `--force` can be specified on a per-package basis to help with these problems.

Some other options are also available, for example:

- Ignore the RPM; do not attempt to delete or update it (`:i`). This is useful if an RPM has been installed manually (perhaps for testing) and should not be deleted by the automatic update.
- Schedule a reboot if this RPM is changed(`:r`). This is useful for important updates such as new kernel, which demand a reboot.

4.2.2 rpmsubmit

`updaterpms` obtains the the copies of the RPMs for installation from an NFS-mounted repository. This repository is replicated on several servers by an LCFG subsystem script which synchronises the copies before the nightly update.

A program called `rpmsubmit` allows authorised users to submit new RPMs into the master repository. This is capable of insisting on valid PGP signatures, and ensuring that corresponding source RPMs are available for all submitted RPMs. Currently, `rpmsubmit` uses NFS to copy the files into the repository, but this is not ideal because signature verification occurs on the submitting client; we would like to rewrite this an a client-server application.

5 LCFG in Action

This section shows how LCFG is used in practice during a number of common tasks; most normal client installations and system rebuilds are performed by junior technical staff:

5.1 Installing New Machines

A new host is installed by first creating an LCFG file specifying the configuration. Frequently, this contains only included classes and, if the machine is intended to be identical to an existing one, then the LCFG file for the existing machine can simply be copied. This example shows a typical configuration for a student laboratory machine:

```
#include <linuxdef.h>
#include <linux_rh62.h>
#include <linux_wire_at1.h>
#include <linux_cs1.h>
#include <dell_optiplex_g1.h>
```

The included classes define the machine as a Linux system, running Redhat 6.2, using servers on the Ethernet segment AT1, configured as a standard first year Computer Science undergraduate machine, and based on Dell G1 hardware.

At present, the Ethernet and IP addresses are entered separately into the NIS and DNS tables, however, there is no reason why these values could not be generated from the LCFG configuration file. The machine is then booted from an install floppy [3], and after a single warning prompt, it performs a full unattended install.

[3]Portables normally require a CD rather than a floppy because of the extra drivers required for the PCMCIA

5.2 Customising Machines

Individual machines can be customised simply by overriding default resources in the LCFG file. For example, to increase the logging level for VMware:

```
vmware.loglevel 3
```

Servers typically have more machine-specific resources than normal clients, but these are rarely more than about one page; groups of resources which perform a related function are usually collected together into a separate class file.

5.3 Rebuilding Machines

If a machine requires a rebuild after an OS corruption or hardware failure, it can be completely rebuilt simply be booting off the installation floppy. This is all that is required, even if hardware has been replaced, and even if the machine has been customised to a non-standard configuration.

5.4 Changing Server Configurations

If the class hierarchy is well constructed, it is straightforward to change all dependent client configurations automatically at the same time as a server configuration is changed. Because the configuration of the whole site is held in a single place, we can also identify possible problems in advance.

For example, at our site, the default DNS servers are set by a class file which depends on the Ethernet segment. If we want to remove a DNS server from a segment, we can simply remove it from the class file and replace it with another machine. This will be detected by all clients next time they reconfigure.

It is also very simple to inspect the DNS servers being used by all the clients. This allows us to check that our old DNS server is no longer in use, before physically removing it. One advantage of the crude NIS implementation of resource maps, is that people can simply type:

```
ypcat -k lcfg |grep dns.servers
```

5.5 Changing Security Policies

Many security-related parameters can be set by LCFG resources. Setting these in the appropriate class file allows the security configuration of whole groups of hosts to be manipulated. For example, we could control the ability to access all first year undergraduate machines from a remote ssh by setting the following in the appropriate class file:

```
inet.allow_sshd  ALL : rfc931
```

We are aware that this depends heavily on the security of the LCFG system itself which is currently not as strong as we would like. This is one area being addressed in the current re-implementation. Similar issues involving automatic configuration of security parameters are discussed in [8].

5.6 Upgrading Software

New software packages can be added simply by installing the RPMS into the central repository and adding them to the appropriate configuration file. They will then be installed onto all the corresponding machines overnight. Upgrading a package usually involves no more than copying the new version of the RPM into the repository (assuming the specification contains a wildcard).

To upgrade the operating system, a new set of base RPMs and an install floppy are needed. Once these have been prepared, hosts can be updated simply be changing the LCFG file to refer to the new class file and rebuilding the system by booting off the install floppy. The host will then rebuild with the new OS, but retaining any customised configuration previously in use. Changes in the operating system itself may require changes to some of the resources, however, the abstract nature of the resource means that such changes can often be avoided by changing the way in which the subsystem script interprets the resource.

5.7 Adding a New Subsystem

The modular nature of the LCFG scripts means that it is very easy to add a script to control a new subsystem, and this will often start with a copy of an existing script. The script should use the provided routines to load the required resources into shell variables. It might then start a daemon with this configuration, or it might simply generate a configuration file and allow something else to start any daemon (See example 8). The script itself would then be loaded onto all the necessary machines by including its RPM in the appropriate configurations, and it would be immediately available for use. Adding the subsystem to the boot resources would cause it to start automatically on the corresponding machines:

```
boot.services ANDALSO(myservice)
myservice.key value
myservice.key value
```

6 Future Plans

The basic concepts behind the LCFG system have proven very sound and the system has been extremely successful. However, a number of aspects of the implementation were not intended for wider large scale use. In particular:

- We would like a finer grained access control on the resource "database", so that we could delegate management more easily.
- We would like a more secure and efficient technology than NIS for distributing resource maps.
- We intend to rewrite the framework for constructing subsystem scripts using a more object-oriented approach, and a different language.

It seems likely that we will use LDAP as the configuration resource repository, and a Perl framework for the subsystem scripts.

We have also learned a good deal about the way in which sysadmins want to specify configurations and classes, and we intend to implement a custom language for describing machine configurations. The design of this language is still under discussion, but we would like to provide:

- Multiple inheritance and mutation.
- Some form of typing to allowing better validation for resource values.
- The ability to specify components (such as a disk configuration) which could be used by several machines (or several times by the same machine).

The current system is also tied closely to other local procedures, making it difficult to export, and we would like to address this, so that we can export it as open source (see below).

7 Availability

The actual amount of code in the LCFG system is comparatively small and we believe that the concepts are more significant than the implementation. However, we do intend to make the software available, and we would like to see it adopted more widely. This includes a number of components:

- The code which takes the LCFG files and transforms them into a single resource map.
- The common subroutines used by the subsystem scripts.
- The subsystem scripts themselves.
- The install subsystem.
- `updaterpms`.

As discussed above, the first four of these are currently being redesigned and we intend to release the new versions as soon as they are available. The current version of `updaterpms` is available via `http://www.dcs.ed.ac.uk/~ajs/`.

References

[1] Paul Anderson. System configuration and installation (SANS97). `http://www.dcs.ed.ac.uk/home/-paul/publications/Config.pdf`.

[2] Paul Anderson. Managing program binaries in a heterogeneous unix network. In *Proceedings of the 5th Large Installations Systems Administration (LISA) Conference*, pages 1–9, Berkeley, CA, 1991. Usenix. `http://www.dcs.ed.ac.uk/home/-paul/publications/LISA5_Paper.pdf`.

[3] Paul Anderson. Towards a high-level machine configuration system. In *Proceedings of the 8th Large Installations Systems Administration (LISA) Conference*, pages 19–26, Berkeley, CA, 1994. Usenix. `http://www.dcs.ed.ac.uk/home/-paul/publications/LISA8_Paper.pdf`.

[4] Edward C Bailey. *Maximum RPM*. Redhat Software Inc. `http://www.rpmdp.org/rpmbook/`.

[5] Kirk Bauer. autoRPM. `http://www.kaybee.org/~kirk/html/-linux.html`.

[6] Mark Burgess. cfengine. `http://www.iu.hioslo.no/cfengine/`.

[7] Mark Burgess. Computer immunology. In *Proceedings of the 12th Large Installations Systems Administration (LISA) Conference*, page 283, Berkeley, CA, 1998. Usenix.

[8] Mark Burgess. Managing network security with cfengine. *Login;*, pages 26–28, August 1999.

[9] Caldera. COAS.
`http://linux.davecentral.com/-3724_sysutiladmin.html`.

[10] Yellog Dog. YUP.
`http://devel.yellowdoglinux.com/-rp_yup.shtml`.

[11] Ken Estes. rpmsync.
`http://www.moongroup.com/RPM/00-05/msg00092.html`.

[12] Martin Hamilton.
RedHat Linux Kickstart HOWTO.
`http://metalab.unc.edu/pub/-Linux/docs/HOWTO/other-formats/-html_single/KickStart-HOWTO.html`.

[13] Dirk Lutzebaeck. freshrpms.
`http://rpmfind.net/linux/RPM/-contrib/noarch/noarch/freshrpms-0.7.3-1.noarch.html`.

[14] Scott McDermott.
Introduction to Solaris Jumpstart.
`http://www.octaldream.com/scottm/-talks/jsintro/jsintro.htm`.

[15] Sun Microsystems. Automatic installation. In *Solaris 2.3 system configuration and installation guide*. 1993.

[16] Nils Philippsen. RACE.
`http://www-stud.fht-esslingen.de/-race/`.

[17] Solucorp. Linuxconf.
`http://www.solucorp.qc.ca/-linuxconf/`.

8 Example: The Start Method of a VMware Script

```
# Fetch resource values
LoadResources subnet workgroup smblog external \
              printer encrypt loglevel external

# Allow external Samba connections?
if [ "$external" == "yes" ]; then
  nosa=";"
  hostname=`hostname`
  subnet=`grep $hostname /etc/hosts |awk -F. '{print $1"."$2"."$3}'`
  xinterface="$subnet.0/255.255.255.0"
else
  nosa=""
  xinterface=""
fi

# Create configuration
sed <$smb_conf.tmpl >$smb_conf \
          -e "s@%SUBNET%@$subnet@g" \
          -e "s@%WORKGROUP%@$workgroup@g" \
          -e "s@%SMBLOG%@$smblog@g" \
          -e "s@%PRINTER%@$printer@g" \
          -e "s@%ENCRYPT%@$encrypt@g" \
          -e "s@%LOGLEVEL%@$loglevel@g" \
          -e "s@%NOSA%@$nosa@g" \
          -e "s@%XINTERFACE%@ $xinterface@g"

# Start VMware
/etc/rc.d/init.d/vmware start >>$logfile
```

Building a self-contained auto-configuring Linux system on an iso9660 filesystem

Klaus Knopper <*knoppix@knopper.net*>
http://www.knopper.net/knoppix/

Abstract

Bootable CD-Roms with a small Linux rescue system in business card size [1, 2] or regular size live demonstration CDs [3] are becoming popular recently. Also, some of the commercial Linux distributors as well as non-profit Open Source groups are developing self-running demos that are preconfigured for certain hardware, or contain a configuration frontend. *Knoppix* (Knopper's *nix) is an attempt to not only create a fully featured rescue/demo system on a single CD, but also to unburden the user from the task of hardware identification and configuration of drivers, devices and X11 for his or her specific hardware. The resulting product is supposed to be a platform CD with a stable GNU/Linux base system, that can be used to customize static installations for a specific purpose.

Goal: Creating a fully functional and usable Linux system running completely from a single CD

A frequently asked question asked by people who "just want to have a glance" on Linux to check out how useful it could be for them is, "How can I test Linux without having to change anything on my computer?". Another issue often requested by those already familiar with Linux is, that there is seldom a fully installed and configured Linux system in reach when you could need one for network debugging purposes or simple tasks like converting files from and into different formats, recover lost data from a corrupt file system or run software that simply only exists for Linux. A "portable Linux allround system", but without having to carry around a notebook or mobile computer which can be lost or damaged, wouldn't that be a great help?

Knoppix is a one-CD live filesystem that can be customized as rescue system, security scanner or platform for presentations and demos, or as full-featured portable production platform with tools like KOffice or StarOffice™.

The underlying GNU/Linux base system is modified to boot non-interactively into runlevel 5 with a working X-Window and KDE [6] configuration, with all auto-detectable devices configured, ready to (auto-)start applications.

Reducing space limitations by compression

The core system of about 200 MB (uncompressed) is currently based on the popular RedHat [4] distribution and contains all basic commands and tools for a generic Linux system. That leaves, on a standard 650 MB CD-Rom, over 400 MB for custom applications, which can simply be installed with standard RPM packages on the CD-Rom install/preparation system.

As of Version 1.2, *Knoppix* features a transparently decompressing loopback-blockdevice derived from Paul 'Rusty' Russel's `cloop` kernel module hack. For a standard Linux installation, this reduces the space needed on the CD to about 50% down to 25% of the original filesystem size and leaves more space to custom applications or multimedia datafiles. The compressed live-filesystem is therefore present as a single file on the CD which is being mounted via cloop from the bootfloppy or El Torito bootimage at system startup, from the ramdisk containing the root filesystem. For performance and stability reasons, iso9660 has also been chosen as the underlying filesystem for the compressed image instead of a read-only ext2 filesystem that is common on other live CD-Roms.

The compressed filesystem not only adds free space on the CD, but also reduces access time and head movement of the CD-Rom drive, but handles physical read errors more ungraciously than an uncompressed filesystem and increases production time of a new release, because the whole installation filesystem needs to be compressed before the new version can be burned and tested.

Platform and Applications

Knoppix provides a ready-to-run operating system environment to:

- start security and auditing tools like *nmap* [8], *nessus* [7], *dsniff* and alike. Since there is no permanent storage present on a read-only CD-Rom, no sensitive information can be written or exposed accidentially. The security checks can be performed on computers directly within a customers network by simply booting from the CD on a machine that is already connected to the internal network,
- produce game and application demos that run in a safe and tested environment,
- have a stable demo installation of GNU/Linux available for presentation at trade shows or consulting talks with customers,
- build a customized, read-only Linux installation for educational environment which is preconfigured for internet access and contains all commonly used applications for this purpose,
- present the features and use of GNU/Linux without having to go through a long and maybe complicated installation and configuration process,
- feature a complete rescue and crash recovery system for all kinds of emergency issues with all necessary filesystems in the kernel, and repair tools available.

Technical Details: Boot process, automatic hardware detection and configuration, autostart of X11 and applications

In stage 1 of the boot process, the Linux loader LILO from the boot section of the el torito [5] 1.44 MB floppy image on the CD-Rom tries to read the kernel (currently 2.2.16) and an 4 MB compressed initial ramdisk. The size of this initial ramdisk determines the minimal amount of memory needed to use the distribution.

Without XFree [9] and KDE, about 8-16 MB of RAM seem to be sufficient for a working textmode-only environment.

In stage 2, the boot ramdisk tries to autoprobe for the most common SCSI adapters and identifies the CD-Rom drive where the *Knoppix* CD is located. The minirootdisk features a statically linked shell with commands like `mount` built in, since the space on the bootfloppy is limited. For compatibility reasons with current floppy drives, only a 1.44 MB floppy image is used on the CD instead of a 2.88 MB. The boot script tries to find the

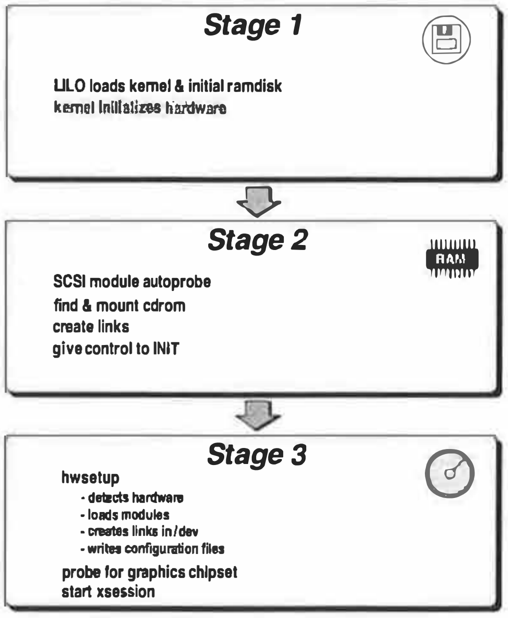


Figure 1: The boot process

Knoppix CD by mounting all CD-Rom drives and checking for a directory `KNOPPIX` that may contain a directory tree for the root filesystem or a file with the same name containing a compressed iso9660 image of the file system which is then mounted via the `cloop` device. If no CD is found, an attempt is made to find the `KNOPPIX` directory on an existing ext2 harddisk partition, containing a complete installation tree.

In either case, symbolic links are set to the uncompressed (or transparently decompressed) directory tree.

After the CD has been mounted, dynamic library cache and paths are initialized and space is freed on the root ramdisk by removing files that are no longer needed for the setup process.

If a swap partition is detected during device/partition scan, an attempt is made to utilize it via `swapon` to increase workspace for applications.

Also in this stage of the system startup, additional ramdisks are mounted with a writable ext2 filesystem for `/home` and `/var`. Their size is adapted from the available amount of real memory. Symbolic links to system directories are created and control is given to `init`.

In stage 3 of the boot process, `init` calls a finalizing setup script named `sysinit`. In this script, the automatic (or, if "expert" mode was selected, manual) hardware setup is done. `hwsetup` - a selfmade tool that uses the `kudzu`-library [4] - detects devices, loads all necessary driver modules for known hardware, sets up symbolic links in `/dev` and writes configuration parameters and options to the corresponding files in `/etc/sysconfig/` on the ramdisk. Parameters that cannot be auto-detected (frequency ranges of old monitors, desired keyboard layout, language) are assumed with reasonable defaults. A single X-Window session is started if the graphics hardware has been identified correctly. Default is truecolor in a resolution of 1024x786 pixels if possible, or 800x600/640x480 at 8-bit if the graphics adapter cannot handle higher resolutions or color depths. If detected, the accelerated XFree (3.3) drivers are used with specific options depending on the detected graphics adapter.

The KDE Desktop manager (currently Version 2.0 beta 3 as of this writing) is started only if there is at least 30 MB of RAM left after all ramdisks are mounted and all necessary device drivers are loaded. Otherwise, a less memory-consuming windowmanager (like `twm`) is used, if XFree can be started at all.

Network device parameters can be set with a tiny `dialog`-based GUI from within KDE, if needed. For dialup Internet access, `kppp` or `isdn-config` [4] are provided.

Security issues

All user passwords, including the root password, are disabled and locked by default. That makes it impossible to log in via network or even on a local console. Therefore, all processes must descend from the shells running on the system console(s) or from the X-Window session that is spawned by `init`.

If the *Knoppix* user wants to enable a remote login, for example via the *openssh* daemon, she has the choice of adding a user with password, or generating an authentication RSA key for this user. There is no `inetd` meta daemon and no standalone servers running per default, that could be accessed from the network, if network is even configured.

A difficult issue is the local security and access to devices, because there is no reliable way to protect the switching from an unpriviledged user account to the system administrators id if passwords have been disabled. Having "dummy passwords" or default crypto keys that are written on every CD would of course breech security. In the current version, the automatically logged-in user at the system console is the system administrator. This can be changed in `inittab` and in the `xsession` init script at the preparation phase of a new CD-Rom image; for example, if the resulting system is supposed to work as a game or demo plattform rather than as recovery system or security scanner. Games, user-mode applications or demos should run fine with an unpriviledged account. In this user-only mode, runtime configuration of priviledged drivers, devices and configuration files is impossible.

Project status and distribution policy

The *Knoppix* CD currently contains the base linux system software plus utilities for a rescue system, current security scanners, demos and some of the KDE and Gnome games. It boots and auto-configures correctly on most ix86 comptible desktop machines. The most common PCI cards (network, sound, SCSI) are auto-detected, drivers are loaded and mice, modems, CD-Rom devices, CD-Writers etc. are given their appropriate links in the `/dev` directory. Harddisk partitions are detected and corresponding icons are added on the KDE desktop.

Most problem cases where the automatic hardware-detection fails have been Notebooks with new chipsets that are not supported by the current linux kernel yet.

In case of failure to correctly auto-detect and configure all necessary hardware components, the CD can be booted with an "expert" option for interactive setup.

The *Knoppix* base system (excluding vendor-specific add-ons which are separate products) is an Open Source project and covered by the GNU General Public License Version 2. The program sources of the software included is available from the vendors specified in the RPMs, the sources of the *Knoppix* setup tools or patches for programs needed to generate the automatic setup system can be found at [10], if not already present on the CD-Rom.

Because of legal issues (i.e. US crypto export regulations and software patents that may disrupt commercial and noncommercial use of software included on the CD, that is otherwise covered by a Free Software license, for example strong crypto versions of KDE or other SSL-enabled software), the author does not currently distribute the *Knoppix* CD publically. Individuals or companies can order a customized CD version for evaluation or software projects directly from the author.

Features in development and (previous) TODOs, sorted by priority

1. Use a transparently uncompressing block device to increase the space available for installed programs. *Status*: Done. Needs cleanup and more checks.
2. Disable spin-down of CD-Rom Drive for faster CD-Rom access after periods of idle time. *Status*: Still looking for a way that works vendor-independent.
3. Clean up boot disk (only 50 bytes or so left), make `insmod` a builtin function of the initial shell. *Status*: `insmod`, `rmmod` and other commands are now builin part of the initial shell. Most SCSI drivers of Kernel 2.2.16 fit on the 1.44 MB boot-floppy now.

4. Generate set-ups that are easily customized for different purposes rather than install and uninstall RPM packets before creating a new image. *Status*: Still working on a method to do this easily.
5. Enhance interactive setup for using existing swap partitions. *Status*: Existing and previously initialized swap partitions are now detected and utilized during the boot process.
6. Add session configuration that can be customized, which can be saved on removeable media. *Status*: Not done yet.
7. Enhance the auto-detection process, increase the number of known cards and recognized graphics adapters. *Status*: Used information retrieved from betatesters to create configurations for problematic hardware (mostly Laptop chipsets), ongoing process.
8. Add support for automatic, non-interactive setup of isapnp devices. *Status*: Not done yet.

References

[1] `http://www.innominate.de/`

[2] `http://www.linuxcare.com/`

[3] `http://www.demolinux.org/`

[4] `http://www.redhat.com/`

[5] `http://www.ptltd.com/products/specs-cdrom.pdf`

[6] `http://www.kde.org/`

[7] `http://www.nessus.org/`

[8] `http://www.nmap.org/`

[9] `http://www.xfree.org/`

[10] `http://www.knopper.net/knoppix/`

About the author:

Klaus Knopper holds a master's degree in electrical engineering and works as a consultant and programmer for Linux-based intranet/internet gateway solutions, firewalls and VPNs.

Introducing Linux File Services into a Windows NT Network: A Pilot Program

Richard R. Morgan

Consultant – VistaRMS, Inc.

Abstract

This paper is a report on a pilot program to implement Linux and Samba as a file server in a telecommunications company of about 5,000 people. The intent of the pilot program was to demonstrate the feasibility and cost–effectiveness of using Linux servers in the enterprise, as well as to illustrate the seamless integration with Windows NT desktop clients that Samba can provide. We will review the computing environment, the reasons for certain choices in hardware and software, as well as the steps in the implementation process.

Introduction

Linux seems to be on everyone's mind today. My workgroup is no different, except that unlike many of the pundits and predictors of success or doom, we are happily using Linux daily.

As a long–term consultant to a company in the telecommunications industry, I have the opportunity to use Linux every day in my work. In addition to our many other projects, my group tests the latest systems and software with Linux, helping to set the direction of information technology for the company.

So, we were duly excited when we got the chance to create a pilot program to use Linux as a file server platform. We knew it would be a challenge, though the process seemed straightforward. But, as the old adage goes, "the devil is in the details." The challenges weren't all in the software or technical realm; many of them sprang from getting interdepartmental cooperation and interaction with our user base.

Project Definition

Environment

Servers – The existing file servers throughout the company are predominantly based on Windows NT 4.0. They are a wide array of Hewlett–Packard servers, such as the LH–Pro, LH–3, and LH–4 series. Elsewhere in the company, there are numerous other Unix servers from Sun, Hewlett–Packard, and IBM. So, the use of a Unix (or 'Unix–like') solution was not a completely foreign idea, although these Unix servers typically run large databases and applications. This meant our local desktop administrators often had little experience in this area.

Desktops – The client machines throughout the company are also based on Windows NT. The standard desktop machine is complemented by a relatively high proportion of laptop machines, also running Windows NT. All users have home directory space provided to them on file servers for storage.

Politics – It is a fairly safe to say that Windows NT is an entrenched platform, with a whole organization built up to support it. However, throughout the project, we have run into interested participants who have shown a great interest in Linux and offered their help and enthusiasm for its introduction.

The Problem

What's the problem? Why not just stick with Windows NT as a file server platform? All is not perfect in that arena. Many servers are running out of disk space and others are simply overburdened with traffic and numerous Windows NT services and need relief. The company continues to grow rapidly and the server burden will not diminish soon.

On the licensing side, the Microsoft model of charging for Windows NT clients and NT server licenses seems standard enough. However, the separate and additional cost of licensing every single desktop machine for 'client access' to use NT–based file and print services adds greatly to the expenses of the company.

The Solution

A potential solution to these problems is the introduction of Linux servers into the network. Immediately, client access license costs can be eliminated by using Linux (which has a licensing cost of $0) to provide file and print services. Other costs such as retraining may be incurred though, at least initially.

Anecdotally at least, Linux seems to make better use of hardware. So, our theory was that NT–based services, including domain control (BDC), Server Management (SMS), and others such as WINS could be moved to smaller, less expensive machines. We would then install Linux on the existing file servers, thus extending their lives. The configurability of Linux would allow us to greatly customize the installation and provide a file server with a minimal operating system footprint.

Linux could breathe some life back into machines that were overburdened due to the plethora of Windows NT services being run on them, in addition to serving files and managing print jobs.

But, we had to prove all of this first. Thus, our pilot program emerged.

Linux – So, Linux is the operating system we decided to try. But which Linux? Hewlett–Packard made the choice for us, really. Throughout our offices, all of the file server hardware is HP gear. We have numerous servers, most with warranties and paid–up support contracts from HP. HP chose to 'certify' Red Hat Linux (starting with version 6.0) on our categories of servers. We felt we could get technical support from HP, even using Linux, should we ever need it.

We were comfortable with Red Hat Linux because we felt our greatest battle would be education; both at the system administrator level, where NT–trained administrators would need technical knowledge from a wide array of published books and training programs, and at the management level, where the managers would have at least heard of Red Hat and its tremendous IPO.

Technically, we also liked Red Hat 6.1 because it offered drivers for the Hewlett–Packard Mega–Raid cards with the stock distribution, and like previous Red Hat distributions, it offered a straightforward manual installation process and the convenient Red Hat Package Manager (RPM).

Samba – Now, the star of the show: Samba. Samba is a triumph in the Open Source world. Samba is the server software that speaks Microsoft's server message block (smb) protocol. Samba would allow us to communicate with our Windows NT desktops and completely emulate a Windows NT file server.

Project Goals

The goals of our program were determined at the outset, and seem to have remained constant. We intended for this implementation, which would occur at only one building with 200–300 users, to be a true pilot. We would be doing all of this with live users and their data, so a careful approach was required.

We wanted to prove the concept, the technology, and also learn enough to be able to replicate the program throughout the company if the pilot was a success.

We hoped to encounter as many problems as possible during the pilot, so that we would be better prepared for a real, full–scale implementation.

File Server Conversion

Specifics of Tools: Hardware

For the file server, we used a Hewlett Packard LH–3 server. It is a dual processor machine (Pentium–II 450mhz) with a 70GB (available storage, using hardware RAID) disk array and one gigabyte of memory. This server configuration is commonly used throughout the company's field offices. We hoped to implement the Linux solution at field offices nationwide, so we chose the same configuration as in the field.

We partitioned the disk to have a large /home partition for the user home directories.

Specifics of Tools: Software

As mentioned above, we chose a stock version of Red Hat Linux (version 6.1, kernel 2.2.12–20smp).

Samba 2.0.5a was current at the time and had the features we needed. Some newer versions of Samba are available now, so we may eventually upgrade the system when necessary. We started out with a simple and small smb.conf (the Samba configuration file) and added directives only as we fully researched and tested them. After making configuration changes, we used the `testparm` utility provided with Samba to validate the changes.

We chose Secure Shell (SSH version 1.2.27) as a secure method for remotely administering the machine. Our Windows NT administrator installed the TeraTerm client on his NT machine to provide ssh access. Big Brother was used to remotely monitor the machines, and we used Bash shell scripts to script the conversion.

While we didn't know exactly what we might run into later in the project, we trimmed a lot of unnecessary packages from our installation, especially games, editors, documentation, extra window managers, and graphics tools. We chose to leave the C libraries in place, in case we needed to recompile some software. Webmin (version 0.74), our web–based administration tool required the Apache web server, so we installed that also.

Preparation & User issues

Whose data will be moved? – This area was a major challenge. The move to the new file server coincided with the physical relocation of about 300 users from three different buildings around our metro area over a two–week period. This meant that we had to coordinate well with our facilities personnel to know exactly who would be moving, and on what dates.

How much data will be moved? – The amount of data was directly related to whose data we would be moving. Some users generate very little home directory data, but others, such as developers and graphic designers can have huge directories. Even for regular users, those gigantic Excel and PowerPoint files can take a lot of room.

The amount of data being moved affected the time it took to transfer the files across the network, as well as our choices in partitioning the server disk space. The final toll was about 5.5GB of files from three servers.

How will users be authenticated? – This was relatively easy. Our network authentication is done via the Microsoft NT domain controller model (PDC/BDC). Samba made it very simple to point requests to a particular set of BDC servers for user authentication. This is the `security = domain` directive in the smb.conf configuration file. Along with that directive, you must set the `password server` directive to point authentication requests at the correct machines.

Remember to withdraw access to old file server after conversion – To prevent confusion and data corruption, we did the conversion during off–hours, and after we converted the user files, we removed the user's access to the Windows NT server. We first set the permissions to read–only, with the NT administrator as owner of the files, to help answer questions about the conversion. Later (about two months), we removed all access to the user's home directory on the old server.

Handle automatic redirects in Windows NT User Profile – When our Windows NT users login, a login profile sets their 'H' drive to the correct file server with their resepctive home directories. After the conversion, we ran through the profiles of the converted users and changed this mapping. Their 'H' drive would still point to their home directory, except it would now be on the Linux box.

Handle user's manual drive mappings via email notice and Help Desk support – Some users had manual drive letter mappings to the old Windows NT servers. We sent out several emails to the affected users warning them before the conversion, and also reminded them after the conversion of the need to change any manual mappings. We also enlisted the support of our internal Help Desk in answering questions about this change.

The Conversion Process

Set up users on Linux system using shell scripts – Once we had a list of the users, we fed this list to a Bash shell script that created the user accounts on the Linux machine. Their shells were set to `/bin/false` to disallow direct logins and no passwords were established.

Create user home directories using shell scripts – The same script, which simply wrapped the standard `adduser` script, created the user home directories.

Control share access – To prevent anyone from poking around in the Linux machine via Samba shares during the conversion process, we chose to limit the access to all shares to the root user. This was done in the Samba configuration file, smb.conf. After the conversion, we removed this entry to allow full access to users.

Move the data into a holding area – On the Windows NT side, we completed copying the data from each server before proceeding, in case there was some failure or other problem (the principle of "atomicity"). We wrote scripts on NT that copied the contents of the home directories of the affected users. These contents were copied to a holding area on the Linux server. We logged all files copied.

Verify the copied data – As the data from each Windows NT server was copied to the Linux server, we checked both servers to verify that all files were copied. We checked file sizes for files and directories and did a spot check of some files by opening them in their respective applications.

Move data from holding area to user home

directories – Once we were sure that we had the correct data set, we copied the data from the holding area into the proper user home directories using a relatively simple `cp` command.

Modify file permissions and ownerships using shell scripts – During the copy process, the converted files were written with the root user as owner. We wrote a script to recursively move through the user's directories and set the proper permissions and ownership on files and directories.

Review logs and handle the exceptions and errors – All the scripts we used created log files of their actions and any errors encountered. We reviewed the log files throughout the process (`tail -f` *`logfile_name`* is rather useful for this.) After the conversion, we manually researched and resolved the errors, which were very few.

Test user logins and read/write access of Samba shares – The final step was to physically log in as several different users and confirm that the proper drive mappings were being set, and that the user could properly access his files. We tested both reading and writing to the directories as well as reading and writing of converted files.

Post–conversion tasks

Enable space management: Set up filesystem quotas – The existing Windows NT servers use a third–party tool, SpaceGuard, to control disk space usage. Each user is given a 30 megabyte limit on disk usage. This function was easy to provide in Linux using filesystem quotas.

The basic quota system on Linux is established and administered for users on a filesystem basis, so we have a dedicated /home partition on the Linux machine. Each user has a hard limit of 30 megabytes. There were many exceptions and after a bit of training, these were left to the Windows NT administrator to handle.

Before enabling quotas, we created a script to run through user directories, calculate their sizes, and report the worst offenders. This helped us know what to expect from users when quotas were turned on, this time with hard limits.

Enable secure access: Set up Secure Shell and disable telnet – To help secure the box, we disabled ftp, telnet, and other services. We installed Secure Shell and trained the administrator to use ssh and scp when administering the box remotely. The machine is behind our corporate firewall, but Windows NT administrators occasionally dial in remotely to fix problems.

Enable remote administration: Install Webmin – As part of ongoing maintenance, new users must have accounts added to this machine. We installed the Webmin tool to provide Help Desk employees with the necessary facility to add new users. Webmin is extremely customizable and we cut its functionality down to this single function.

Enable remote monitoring: Install the Big Brother client – During this project, we were also installing six Linux print servers. To monitor the whole thing, we chose the Big Brother monitoring tool (version 1.4a). Big Brother is simple to use, extensible, and its graphical alerts were a good fit for our local Windows NT administrator. Be aware though, Big Brother is not covered by the GNU General Public License.

Enable remote syslogging: Send logs to a separate log server machine – We enabled remote system logging. This allowed us to consolidate the logs of many machines on one central box, and then run scripts against the logs for reporting or troubleshooting purposes. We configured `/etc/syslogd.conf` to send logs to a separate logging machine. Rather than using a local destination of `/var/log/messages`, we used a machine name, `@vauas005`.

What We Did Right

We did good testing in our lab. Several weeks ahead of the conversion, we did extensive testing on real data to refine our procedures and scripts. With the cooperation of the local NT administrator, we ran through the conversion process with a set of user directories. With this testing, we found holes in our knowledge of Samba and worked through them. We learned how the BDC system works on Windows NT, and how users are authenticated through Samba.

Our biggest unknown remained the amount of time necessary to move files across our WAN.

We scoured the documentation for tips and gotchas. We bought every Samba book that existed and read through them. The O'Reilly book, Using Samba seemed the most help to us, especially the 'fault tree' section. It helped immensely in focusing our troubleshooting efforts. We also reviewed the documentation provided with Samba and used it to create our preliminary conversion checklist.

We were very careful and methodical in planning the conversion of the data. With the confusion of so many people moving, we had a hard time knowing exactly what to convert. We worked with the facilities group closely to get the correct list. We even pushed the conversion back twice, knowing that we didn't have the correct roster.

Several times before we converted, we talked through the conversion process, making a list or a kind of 'script' of who would do what tasks and the schedule for each conversion item. This really helped keep everyone on track during the conversion.

We wrote good scripts that did most of our work, checked for errors and logged all transactions. No one wants to do lots of manual work. It is boring and an avenue for errors to creep into the process. To help speed the process, we wrote a number of Bash shell scripts and Windows batch files to automate the conversion.

Each script had options for creating logs of its action, which is especially important if the script changed data. We were able to run each script, having it echo its actions to the screen while testing so we could understand the impact on a test system before we actually changed or moved any real data.

During and after the conversion, we reviewed our logs looking for problems and errors. We had volumes of logs, which tracked every action taken.

We were able to use lots of open source software. This is my favorite part. We used Linux, Samba, Big Brother, quota, and the Bash shell to quickly and reliably accomplish our goals with no licensing costs or other hassles.

What We Could Have Done Better

As a pilot project to be used by real users, this project was bound by most of the same requirements applied to our production servers. Eager to get started, we did not do a very good job of gathering formal requirements and getting inter–departmental agreement on them. Thus, during the final days before the conversion, we still had requests for certain functions coming in from other groups.

We also did not formulate a solid exit strategy for our group. This led to confusion among the groups about who was to manage the server immediately after the conversion. We assumed, wrongly of course, that since the project was a pilot, we would continue to run it and maintain the machine.

To remedy these items, we could have used better communications. We did create an intranet website for the project but assigned no single person as the maintainer. The site did not get updated in a timely manner or even very often.

Even with all of our testing, during the copy process, where user files were copied from their original locations on the Windows NT servers to the 'holding area' on the Linux server, we had a small slip–up. As the files were copied, the file dates ('last modified') were all set to April 21, 2000 (the day of the conversion), and reflected the current time. We should have copied the files in a manner that would have maintained the correct dates on the files, perhaps by bundling them with zip, tar, or a similar tool. We have had no users complain, though.

As pure techies, we were all more interested in getting the project done and the machine operating well. We didn't spend the necessary time doing a strong business case and things like cost–benefit justifications. I really feel that the lack of this has hindered our efforts somewhat.

Future Plans

We are still monitoring the performance of this server, proactively looking for problems. We have also implemented Linux and Samba as a print server solution for our building. We were able to use small Hewlett–Packard VL desktop machines as print servers. These machines were actually surplus equipment, no longer useful in the world of the Windows NT GUI.

Our group will now complete documentation which will be the guide to implementing this platform throughout the company. To deploy Linux widely, we will need a more streamlined installation model, perhaps disk imaging or a 'kickstart' style of installation.

We have started receiving additional requests from various groups for Linux–based file and print services. To provide this, we need a better method of managing quotas and groups. We are searching for a directory–based quota system, which is a feature available (albeit through third–party software) to our Windows NT servers now. Managing users on individual servers will grow to become a burden quickly, so we would like to manage groups and access centrally. For this, we are evaluating NIS as a compliment to the Microsoft PDC/BDC model.

Company management needs to set the course and fund the Linux training of Windows NT administrators. We investigated training options and found that there is a great deal of Red Hat training being conducted nationwide. Companies like LinuxCare can even provide support for Linux machines.

Our group now goes on to explore the applicability of using Linux and other Open Source tools elsewhere in the enterprise, particularly using LDAP and clustering technologies.

Conclusions

Plan well, test a lot, and work with other groups to get full cooperation. Everyone should understand the role of the new server and his/her role in the process of conversion.

Take a long–term view, as Linux is really just beginning to become popular (beyond media hype) in the corporate world.

Acknowledgments

None of this project could have been completed without the tireless and imaginative work of Rob Fike, Jim Edwards, Frank Hum, Geoff Silver, Susan Maitra, P.S. Ramesh, and Roger Maduro. Thanks to all of you.

We would like to thank the late Bob McLaughlin for having a long–term architectural vision that included Linux and for being tough enough to promote that vision well before Linux became *The Next Big Thing*.

Author Information

Richard R. Morgan is a consultant with VistaRMS, Inc., in Herndon, Virginia and has a range of experience that includes system integration, web design, and software development.

He began his computing odyssey many years ago by teaching himself BASIC on the Timex–Sinclair 1000. He is the corporate secretary and webmaster for Tux.org (http://www.tux.org) and his local LUG (http://novalug.tux.org).

Richard lives in Haymarket, Virginia, with his wife, Laura, daughter Abby, and all of their pets. He can be reached at rmorgan@tux.org.

References

Blair, John. *Samba: Integrating UNIX and Windows*, SSC, 1998.

Eckstein,Robert, David Collier–Brown, and Peter Kelly. *Using Samba*, O'Reilly & Associates, January 2000.

Frisch, Aeleen. *Essential System Administration,* O'Reilly & Associates, September 1995.

Software

Samba – http://www.samba,org

Big Brother – http://www.bb4.com

Red Hat Linux – http://www.redhat.com

Bash shell – included with Red Hat Linux

Linux quotas – included with Red Hat Linux

Secure Shell – http://www.ssh.fi

Gaining the Middleground: A Linux-based Open Source Middleware Initiative

Dr. Greg Wettstein Ph.D., *North Dakota State University*
Johannes Grosen M.S., *North Dakota State University*

August 25, 2000

Abstract

Central to the development of the Internet, and the resulting business paradigm shifts, has been the adoption of standards which have provided the necessary framework for computer systems to cooperate and exchange information. The next major step necessary to support the continuation of this process is the development of software protocols and tools which provide the ability to manage the delivery of services to users.

This paper describes a Linux-based middleware initiative which has been deployed at North Dakota State University to support a Category of Service information delivery system for the North Dakota Higher Education Computing Network. The system was designed to support a computing model where the majority of the service infrastructure is supplied by Linux servers and which is extensible to the 11 state institutions comprising the HECN.

The paper begins with a discussion of middleware and its importance in the development of next-generation information systems. Included in this discussion are the topics of identification, authentication and authorization and the requirement for them in a modern middleware solution. Central to this topic will be a discussion of the potential threat that proprietary middleware solutions pose to the continued penetration of Open Source efforts into the enterprise computing model.

The paper then discusses in detail the design and architecture of the User Services Management System which implements the middleware solution at NDSU. The three fundamental components discussed will be the User Services DataBase, LDAP meta-directory services and the User Account Management System .

Discussion of the system architecture will include details of how standard services have been modified to operate under middleware control.

The paper will conclude with a discussion of the implications of this work. Also included is a discussion of the next generation of the system architecture.

Introduction

North Dakota State University (NDSU) is one of eleven colleges and universities in the North Dakota University System. With an enrollment of 10,000 students, NDSU is a land-grant institution offering several Phd programs and home to several internationally recognized researchers in the fields of chemistry, engineering, and agriculture. NDSU is a member of Internet2 and a partner in the Great Plains Network, a NSF-funded regional network connecting six mid-western states and supporting earth system sciences research. The authors are employed by the NDSU campus computer center, Information Technology Services, (ITS). Dr. Wettstein is the senior system administrator and Mr. Grosen is the associate director for networking and multi-user computing systems. ITS provides desktop computing support and services, training, networking, and other information technology to the NDSU campus. ITS is also home to the SENDIT network, a K-12 technology services initiative for the state of North Dakota.

As a rural state with a small population, North Dakota has faced and continues to face many challenges in deploying information technology services. In the mid

1970's, university system administrators conceived of the notion of tackling some of these problems by centralizing as many services as possible and deploying a communications infrastructure to provide access. The resulting system of services was named the Higher Education Computing Network (HECN). Initially, the HECN concentrated on providing administrative computing services and the University of North Dakota was designated the administrative computing host site. In the late 70's it was recognized that technology was becoming an important component in academics and, in 1980, NDSU was charged with the responsibility of centrally providing academic computing services. Initially, the infrastructure consisted of a single IBM mainframe computer. Over time, it came to include multiple servers and services such as email, domain name services, USENET news, Unix shell access, etc. When the mainframe system was removed from service four years ago, the model for delivering services remained essentially the same; users accessed services via interactive logins over the state's wide area networks.

As desktop PCs became more ubiquitous and the advent of graphical interfaces made these applications more accessible to the non-technical, users began to prefer using their desktop software over that available on the centralized servers. Concurrently, a survey of HECN users and administrators revealed that there was consensus on the need for a unified electronic "phonebook" or directory. The survey also revealed the desire for a standardized scheme for email addresses. Two criteria were identified: first, the recipient portion of the email address should be based, as closely as possible, on the person's full name and, second, the dependency on hostnames for mail servers should be replaced with DNS domain names and MX records. These observations and requests became input into a major re-design of the infrastructure for providing HECN academic computing services.

During the design process, additional technical requirements were developed. Key among these was moving to a client-server architecture using micro-servers running Linux. The old server infrastructure was based on several Unix hardware and operating system combinations. This requirement would allow us to reduce the heterogeneity in the server infrastructure, simplify management, ease development, and lower costs.

Another technical requirement was to move to a Category of Service (CoS) model. In the "old" model, users were assigned Unix logins which gave them access to all services (email, printing, etc.) supported on the server. Servers were, in effect, equivalent to certain standard sets of services and users were assigned according to their needs. In some cases users were assigned to multiple servers to provide them with all the services they required. In the client-server model, Unix logins could no longer be used. Thus, a new way of creating and assigning sets of services for users was required.

The CoS model was the answer to this problem. Services would no longer be tied to any particular server. Each user would be assigned a single "login" with a single password. Each service would be separate from all other services and enabled or disabled without affecting any other services for that user. Custom sets of services could be created for users without the need for multiple server logins.

Two fundamental problems arose out of these requirements:

- how to identify, authenticate, and authorize access to services
- how to implement and manage the CoS model

These problems led us to the discovery of the importance of *middleware* as a basic component of the IT architecture we were developing.

The Necessity for Middleware

The requirements and design process described above began in 1997. At that time, the notion of middleware was not widely known in the field of IT. Instead, middleware was understood by its major components: *identification*, *authentication*, and *authorization* (IAA). These components must be clearly understood before the importance and role of middleware can be understood.

Identification, Authentication and Authorization

A key component in a distributed, client-server-based network is the ability to reliably and consistently identify users. It is extremely desirable (at least from a user's perspective) if there is a single identifier that universally identifies them throughout the enterprise. This latter goal has typically been very difficult to implement because, in a heterogenous network, servers and/or services may have different requirements for an identifier. In addition, in larger enterprises, the network may be composed of several administrative subdomains making consistency an administrative challenge at the very least. Ideally, there should be a single external identifier (user id) and some means of mapping that identifier, if necessary, to an internal, invariant identifier. This solves the problem of the heterogenous environment and permits the user id to be changed by the user without affecting the internal identifier.

Once we have the means to identify users the next challenge is to verify that the identity presented is being used by the person it is assigned to. This is authentication; the process of identification verification. Traditionally, authentication has been accomplished by the use of an identifier and password pair. The user id uniquely identifies the user to the system and the password can be used to verify the identity of the user. As with the identifier, it is desirable to have a single password that can authenticate the identifier throughout the enterprise. It is also important that every effort is made to insure that the use of the password on the network is secure from "snooping" and other attacks.

Most discussions stop at this point; we have an identifier and the means to authenticate the use of that identifier. However, in this model, with a single identifier and password for every user available to every server on the network, how do we determine whether to allow access to a service? Traditionally, authorization is implied by a valid identifier and password as in a Unix login. If a user can login to the server they have access to whatever services are available on that server. In the distributed computing model, this is no longer enough. We have to be able to determine whether a valid identifier is authorized to access a service and the terms under which the access can occur. Thus, a complete solution is only possible when we have all three components; identification, authentication, and authorization.

The Role of Middleware

Middleware is a term which refers to the set of services composed of IAA, APIs, and management systems which support the needs of a distributed, networked computing environment. As the name "middleware" suggests, it is, using the hierarchical model, a layer of services which sits between applications and the services they access. An excellent overview of middleware can be found in RFC 2768[1]. The RFC's authors admit that even today, middleware is still not a well-defined term but the above definition is consistent with the RFC's and is sufficient for the purposes of this paper.

Middleware, at first glance, seems somewhat innocuous or invisible, especially when it works well. This tends to disguise the importance of this layer, however. Middleware is a critical component in the CoS model. It is the middleware which supports the creation and removal of users, provides the ability to verify an identity, manages the addition and removal of services associated with user ids, and even the addition and removal of services available to users. Within the enterprise, a good middleware solution makes the delivery of services transparent to users. There is no need to know which server is providing a given service and a single login/password provides access to all of the enterprise's services, subject to whatever authorization policies have been established. In the business-to-business world, where multiple enterprises establish electronic relationships, middleware will perform the same function but it is complicated by the fact that there are currently no middleware standards in place.

The IT community today certainly recognizes the importance of middleware. The Internet2 community has formed a working group to investigate middleware. Companies like Microsoft (Active Directory) and Novell (NDS) have committed huge amounts of money and, in effect, bet their corporate futures on gaining adoption of their middleware implementations as standards. This makes middleware, in the authors' opinion, the single most important component in future networked computing environments. The Open Source community, and the Linux community in particular, must recognize its importance.

A proprietary solution such as Active Directory or NDS will irreparably harm the Open Source movement. The success of Linux in the enterprise has largely been because of its adherence to open standards and availability of source code. This has permitted managers to easily de-

[1]http://www.ietf.org/rfc/rfc2768.txt?number=2768

ploy Linux into existing enterprise IT architectures where open standards exist. If the middleware standard isn't open the implications are obvious. For example, consider the SMB file protocol and the open-source Samba software. With the introduction of Microsoft Windows 2000 and the incorporation of Active Directory into file and print services, Samba's viability is immediately threatened. This scenario will repeat itself over and over unless the middleware is standards-based and open.

An Open Source Middleware Marriage

The Open-Source community provided the important infrastructure for the development of the CoS middleware solution described by this paper. In the current implementation the three basic components of the IAA process are handled by the LDAP directory server provided by the OpenLDAP[2] project and by the MIT Kerberos[3] authentication system.

The success of this middleware implementation was a function of having access to source based tools which implemented commonly accepted standards and protocols for identification, authentication and authorization of users for information technology services.

LDAP Directory Services.

The most central component of a middleware initiative is the ability to identify a person or entity to which services are being provided. The Lightweight Directory Access Protocol (LDAP) provides both an open protocol as well as an open-source solution for implementing the identification component of the process.

The directory services component of the middleware solution is essentially a database of all objects within an enterprise information delivery system. This notion of managing a computing infrastructure through a network accessible database or directory system is referred to as 'directory enabled computing'. Proper implementations of directory systems provide the advantage of centralizing the tracking of resources as well as decreasing administrative costs by centralizing the control and management of service delivery.

Directory systems are based on representing service entities and computing resources in a hierarchical tree form. The most critical element in the implementation of a directory is the requirement that each user be granted a unique identifier which optimally should be considered immutable throughout the users lifetime in the organization. This identifier forms the basis for the Distinguished Name (DN) in the directory, essentially the equivalent of a database "key." This characteristic is particularly important in the implementation of digital signature systems since digital certificates are branded with the DN of the user which the certificate will authenticate. As discussed in the previous section, this requirement also makes directories ideal for supporting identification in a middleware solution.

Implementation of an enterprise directory system will become increasingly important to the effective delivery of services in a highly networked enterprise computing environment. Decreasing system administration costs is considered to be a fundamental benefit of directory systems. In the future, the role of directories will become critical in implementing strategies such as organization-to-organization partnering and in managing concepts such as organizational role playing. Role playing refers to the notion that an individual may have multiple functional roles in a organization. Often, it will be desirable to vary a user's authorizations based on the role they are assuming at a given time. Directory services will also become a critical component in monitoring and delivering quality of service guarantees and as an integral component in developing policy based computing initiatives.

An enterprise directory system is critical to the successful implementation and deployment of digital signatures based on PKI or X.509v3 certificate technology. Federal initiatives are underway which will require that organizations wishing to do business with government agencies use digital certification to authenticate transactions and decrease administrative costs by reducing paper-based accounting and transaction schemes.

One of the foremost goals in implementing directory services for this middleware project was to provide the mechanism for implementing a single sign-on identifier system for all users. Included in this goal was the desire

[2]http://www.openldap.org/

[3]http://web.mit.edu/kerberos/www/

to unify the email address with the identifier token used to access other services. Implementing this mechanism enables all users to be provided with a single identifier which is not only the preamble for their email address (component in front of the @ sign) but also their login name or identifier for all other services.

The canonical identification token is referred to as the Information IDentification or IID. This token is formed using an underscore to separate the users first and last names. A tie breaking algorithm was developed to insure that the IID was unique across the entire user domain. In addition to the IID each user is granted an invariant identifier which is used to uniquely identify the user throughout the lifetime of the organization. This invariant identifier is used to construct the Kerberos authentication principal and also serves as the machine identifier for the user. The LDAP directory thus provides the mechanism for mapping an external identification element into the identity of a user which in turn provides access to the information elements (attributes) of that user.

This solution offers a number of advantages to the traditional scheme of granting a user a UNIX style userid. First of all the IID is not limited to eight characters which in turn allows the IID to be more expressive of the users actual identity. This, for example, allows email addresses to be constructed which resemble the users actual name. A second and major advantage is that this mapping allows the external representation of the user to be changed without affecting the actual identification of the user within the system. This allows account creation to occur once without the need to modify machine accounts and/or configurations if the user should wish to change their external identification.

The Open-Source middleware solution described by this paper is heavily reliant on directory services to provide not only a repository of user information but also as the central resource for the authorization of services. Classic middleware solutions within the UNIX environment have typically used directories to manage user information such as numeric ID's, personal information (GECOS) and group relationships. A central goal in the development of this solution was to also facilitate the concept of service authorization. Each individual object describing a user entity has an attribute associated with it which describes whether or not the user is authorized to receive a given category of service. In the current implementation a very simplistic methodology is used to authorize services.

Each service category is specified by a tag field within the services attribute. The basic level of services provided by the system consists of authentication services, email access, remote configuration management, dial-up (modem) access and USENET news reading privileges. Absence of an attribute tag indicates that the service is not granted to the user, a lower case tag is used to indicate that the service has been disabled and an uppercase tag denotes that the user is authorized for the service. The following figure demonstrates three users with different states of access to email services:

No email service	services: USENET
Email disabled	services: USENET email
Email enabled	services: USENET EMAIL

This scheme provides support personnel with the means of determining the services granted to any individual user and the status of that access. The ability to disable a service through the directory provides a mechanism for suspending services without disturbing the actual instantiation of a service. For example, with email services, the delivery of mail to the user will continue even if access to the message store is suspended.

Kerberos Authentication

The second pivotal element of an effective middleware solution is the authentication of the users who are identified via the LDAP directory system. The Kerberos authentication system was implemented to provide this component for the CoS information delivery system.

Implementation of an LDAP directory system simply provides a mechanism for identifying a user within the organization and associating the user with a set of attributes describing the user and their role in the organization. An equally important part of this process is the guarantee that the user who is interacting with the system is actually the user identified in the directory system.

In some directory implementations the authentication process is conducted by storing authentication tokens such as passwords as an attribute associated with a user's directory object. Other schemes rely on the use of digital certificates to authenticate the user to the database via the

Secure Socket Layer (SSL) protocol. The Kerberos authentication system was chosen to provide this important functionality for a number of reasons. First and foremost was the desire to separate the authentication and identification components of the middleware process. A second imperative was to leverage the extensive security experience of the Kerberos team in developing a scheme for strong network based authentication. A final rationale was the ongoing concern of the design team with the issue of user credentialling including the imposition of finite time limitations on the authentication and authorization periods.

The Kerberos system is based on the notion of a Trusted Third Party authentication scheme sometimes referred to as a shared secret system. Each entity authenticated by the system is referenced by an alpha-numeric tag known as a principal. A secret key is associated with each principal maintained by the authentication database. A user authenticates to the system by encrypting a request for authentication. Successful decryption of the request validates the user.

Additional security guarantees are provided by insuring that a server providing services to the user is indeed a legitimate server. This functionality prevents security attacks such as IP spoofing and DNS contamination. The authentication server provides the user with a token which is encrypted with a secret known only by the server dispensing a particular service. The server must successfully decrypt this token for the service connection to be properly authenticated.

In addition to authentication the Kerberos system also provided this middleware solution with the resources needed to implement a single-signon system for service access. Web based tools accessed through an SSL secured WEB server provide support and administrative staff a mechanism for managing user passwords. An optimum implementation of Kerberos authentication requires that passwords never be allowed to travel unencrypted on the network. Current client limitations precluded attaining this optimum environment. The use and development of Kerberos provides a solid foundation for developing and strengthening the local security infrastructure as client support matures.

A significant result of this project was the implementation of strong Kerberos authentication and encryption in the Open LDAP directory server. Current implementations of the server include Kerberos IV authentication but did not include support for the current Kerberos 5 release. Ongoing work is being conducted to implement the notion of service classes functioning as authentication entities with respect to service objects within the LDAP directory. This work will be extended as the policy/authorization engine (discussed later) is integrated with identification and authentication services.

The ability of Kerberos V to support separate authentication realms was leveraged heavily in this project. Currently six security realms are supported by this implementation. Support for cross-realm authentication was used to implement inter-operability between servers specific to each organizational unit. The separation of the authentication realms also promoted individual autonomy for the participating organizations. User level administrators within one security realm do not have access to security information in another organization's realm. The separation of the authentication realms also reduces the potential impact in the event of a compromise of one of the Kerberos key management servers.

The ability of the LDAP directory server to identify users merges naturally and symbiotically with the separate authentication realms. Authentication and authorization tools developed as part of this middleware solution use information attributes from the directory server to determine which security realm should be used to authenticate the user. This allows servers supporting separate organizational units to provide services on a cooperative basis with provisioning of services controlled through the directory system.

KerDAP API

The preceding discussion notes the synergy that occurs when elements of identification and authentication are merged. An important component of this middleware solution was the development of an Application Programming Interface (API) which allows common Linux based applications to leverage this middleware-enabled computing architecture. The simplistic programming library developed is referred to as *KerDAP* reflecting the union of the two open source solutions.

In order to be effectively managed through this system, all applications which implement user services needed to be modified to take advantage of the middleware. The goal of the KerDAP API is to encapsulate the mechanics of directory lookups and Kerberos authentication into simple function calls which can be easily integrated into the authentication structure of the open source applications which were used to implement the services.

The current library exports the following four functions:

1. char * IID_Login(char * iid, char *password)
2. char * IID_Service_Login(char *iid, char *password, char *service)
3. char * IID_To_Uid(char *iid)
4. char * Get_Uid(void)

The first function carries out simple identification and authentication when given the canonical identifier (IID) and a password. The second function implements the triad of identification, authentication and authorization (IAA) when given an identifier, a password and a service attribute. In both cases a NULL pointer is returned if the process fails and a character pointer to the machine representation (POSIX uid) on success.

The third and fourth functions are utility functions which are useful when middleware support is enabled in applications. The third function simply carries out the identification process and returns the POSIX userid which the canonical identifier (IID) maps to. The fourth function provides a mechanism for retrieving the value after a successful call of one of the first three functions. All functions cache the POSIX userid in static storage if a mapping is successful.

There are situations where an application needs to be middleware-enabled but source code is unavailable. Other applications implement authentication using an external mechanism. In these cases, the API is of no use and another method is required. KerDAP provides the *kerdauth* command-line utility for these situations. This utility has proven to be particularly useful in a wide variety of CGI applications where middleware support is required. It has also been used as a supplemental authenticator for Squid proxy services as well as user authentication for INND (USENET news). The following usage table summarizes the simplistic character of the application:

Option	Action
-A	Authorize mode
-K	Authenticate mode
-e	IID to authenticate or authorize
-h	Print usage
-s	Service to authorize for
-v	Verbose mode

The *kerdauth* utility operates in either authorization or authentication mode. In both modes the utility reads the user password on standard input. This minimizes the potential for security issues related to passing passwords on the command line. In both modes the canonical user identifier (IID) is passed via the -e switch on the command line. In authorization mode the -s switch is used to specify the service which authorization is requested for.

When used as an external authenticator the *kerdauth* utility returns results via the exit status from its execution. A return code of 0 indicates the authentication or authorization was successful. A non-zero return code is used to indicate a failure condition. Specifying the verbose mode causes the application to print out the results of the authentication or authorization procedure.

Middleware Management - User Services Management System

One of the primary yardsticks for measuring the success of a middleware-enabled computing architecture is an increase in manageability of the information delivery infrastructure. The overall goal of this middleware project was a complete end-to-end solution which provided a seamless architecture for the management and tracking of user services and the ultimate instantiation of the services on a target server.

The multi-component system architecture implementing this solution is referred to as the User Services Management System (USMS). This system is unique in that it provides a complete open-source implementation capable of tracking and managing enterprise information services. Of particular importance is the modular and extensible design which allows the system to expand without organizational or geographical limitations. The following sections discuss the essential components of the implementation.

User Services DataBase

Two strategies exist for the management of service entities within a middleware solution:

1. Canonical directory services.

2. Meta-directory services.

In a canonical directory services implementation all user and service information is tracked in the hierarchical enterprise directory. Classical commercial middleware solutions such as Novell Netware's NDS and Microsoft Active Directory implement this type of solution. A meta-directory implementation uses a separate repository of information and provides a scheme for propagating the data objects into one or more directory servers.

An important initial design decision in this middleware solution was to implement a meta-directory strategy for tracking users and services. Directory server systems such as LDAP tend to be optimized for the rapid retrieval of information elements given a system of filtering constraints or search rules. The most important and fundamental limitation of these systems is that they provide no relational data services nor do they provide important data preservation features such as referential integrity or transaction guarantees. For these reasons the decision was made to implement the management of users and services in a relational database system and to propagate the directory objects through a meta-directory update system.

This approach also offered flexibility for future integration into Enterprise Resource Planning (ERP) and data warehousing projects which are under development by the university system. As was noted previously, middleware management systems will be an essential component of PKI deployments and will need to support and implement the notion of organizational role playing. Integration of service management middleware solutions with administrative and enterprise computing systems will provide useful synergies for implementing a seamless information access and delivery architecture.

The actual implementation of the USDB consists of a relational database and application software that is responsible for populating, updating and managing the system. The relational database component is implemented with Oracle and the application software is written in PERL. A design mandate was to implement the application interface to the relational database with the modular DBI::DBD system. An additional constraint was to implement the database using ANSI standard SQL and datatypes. The overall goal was to isolate the application software from the database implementation so as to allow an alternate database to be 'plugged' in as the backing store.

The database implements a series of tables which track users, services and hosts which implement service delivery. The application layer implements the notion of creating a 'binding' which is a tuple relating a user, service and server. The application software provides an implementation of a rules structure which allows a common service such as EMAIL to be bound to different servers based on parameters such as the organizational unit (OU) of the person receiving the service.

The entire system inter-operates with a mainframe computer system which supports the administrative software systems for the university system. A subset of the user's information is exported from this system to the USDB which provides for essentially real-time updates of the middleware data elements.

A WEB based application is provided for user interaction. This application allows a user to apply for additional services and change various characteristics of their service profile. The system also implements an acceptable use quiz which is mandated by the legal department of the university to insure that users understand and can be held responsible to the information ethics policy of the university system.

LDAP Account Management System

The second major component of the USMS is the meta-directory update system which is referred to as LAMS. This system is responsible for propagating directory updates from USDB to the LDAP directory servers. LAMS also provides a command line interface for making changes in the directory across the master and replicate LDAP servers. The following table summarizes the list of operational modes for LAMS:

Action	Description
Add	Insert a DN
Kill	Remove a DN
Delete	Remove an attribute of a DN
Update	Modify a set of attributes for a DN
Modify	Modify attributes of a series of DN's
Query	Lookup and display a DN

The USDB maintains a set of rules which map particular record fields from the relational databases into various attribute elements for each directory object. When the USDB determines that one of the LDAP exported attributes has changed the LAMS system is called to update the directory object. While LAMS operates on the directory objects and their associated attributes, all requests for update and modification services are done via the user's canonical identification IID. The input for requesting changes is made through ASCII files coded in the Lightweight Directory Interchange Format (LDIF).

All changes are propagated from the USDB into the LDAP directory servers in the amount of real-time afforded by the administrative systems that ultimately serve as the authoritative source of user information. Error messages from the update and replication process are posted back to the administrative team via e-mail so that remedial action can be taken to correct errors. The instances of manual intervention is generally quite low. The most common errors arise from incorrect user reference data which typically requires intervention at the USDB level or higher.

The meta-directory update system can be globally disabled so that updates are not propagated to the master and replicate servers. This feature is useful from a system administrations perspective when there is a desire to hold all the directory servers in a known state. The USDB holds the last modification time for an object as well as the last propagation time. After directory updates are re-enabled all changes to the meta-directory information since the last update are propagated into the directory servers.

The USDB also supports the ability to generate a complete LDAP directory load in LDIF format. This file is suitable for building an entirely new LDAP directory server database representing the current state of information under management by the middleware structure. This feature is useful from a disaster recovery perspective as well as for re-synchronizing all directory servers to a known state.

The low cost of the Intel architecture and the OpenLDAP directory server software makes running multiple directory servers economical which in turn yields important benefits from an administrative perspective. The data center implementing this middleware solution uses one primary LDAP server and two replicates. The LDAP connections are mediated through a fourth server running TCP/IP port redirector software. This provides for load-balancing as well as the ability to remove directory servers from the active service rotation. This feature provides an easy mechanism for synchronizing the directory databases while maintaining a high availability profile for directory services.

User Account Management System

The final component of the services management system is responsible for the instantiation of the service on a server and is known by its acronym UAMS. This subsystem is responsible for serving as the bridge between the USDB and the actual servers providing end-user services.

An important concept in this services oriented solution is the notion of 'binding' a particular service entity to a host. A service entity is most easily understood to be a user with a specific service such as electronic mail. The USDB application layer provides a mechanism where a number of hosts can be defined as candidates for providing a particular service and then placed into a pool. At the time of service creation the binding process selects a candidate server using either a round-robin or weighted averages algorithm and 'binds' the user to that server. The UAMS layer is responsible for taking this binding request from the USDB and carrying out the actions needed to prepare the server to provide the service for the user.

The USDB application layer also supports the notion of so called "host-less" services. These are services which only require KerDAP API services and do not actually require account creation on the server. Two examples of such services are USENET news service and authenticated WEB proxy service. In both cases the *kerdauth* binary is used to provide simple authentication and authorization services via the external authenticator mechanism provided by INND and the Squid proxy server. The only action needed to instantiate the service is the presence of the user in the LDAP directory with an appropriate services tag and a Kerberos principal for authentication.

The UAMS system is designed around the paradigm of queueing requests for user services. Queued requests are stored in separate spooling areas, one for each server. The system provides for locking and arbitration of access to the queue. Requests for service instantiation are stored until the queue is "processed" either by an administrator or by an automated mechanism. Processing of the requests can be carried out either on a per host basis or globally for all hosts which have entries in their request queue. The process of "running" a queue involves reading each request and sending that request via an authenticated and encrypted session to the target host. The results of the requests (success or failure) are appended to the service request and are used to replace the request queue. This mechanism provides for historical accounting of the success or failure of the requests. The UAMS system provides tools for an administrator to review both pending and processed requests.

The actual syntax of the service request are very simplistic. The following tables contain the input for a series of pending requests and the resulting output:

```
imap1.domain:Bull_Dozer:EMAIL:create
imsp1.domain:Bull_Dozer:CONFIG:create
kdc1.domain:Bull_Dozer:KERBEROS:create
imap1.domain:Back_Hoe:EMAIL:delete
imap2.domain:Road_Grader:EMAIL:modify:quota
```

```
imap1.domain:Bull_Dozer:EMAIL:create:OK
imap1.domain:Bull_Dozer:CONFIG:create:OK
imap1.domain:Bull_Dozer:KERBEROS:create:FAILED
imap1.domain:Back_Hoe:EMAIL:delete:OK
imap2.domain:Road_Grader:EMAIL:modify:quota:OK
```

The actual instantiation of the services is carried out by UAMS clients which are installed on the target hosts. The entire UAMS system is implemented with Bourne Shell scripts and relies on the notion of function inheritance to implement the actual service classes.

The basic UAMS client implements a library of common functionality including user account creation via the *useradd* utility. Each service category is implemented by sourcing in a module which has the option of either replacing existing functions with modified versions or using the functionality provided by the base library. This inheritance mechanism also provides the ability to import functionality from host and domain specific service modules. This system allows modules to be written which support enterprises with the need for tailoring services on the basis of organizational unit needs. While the UAMS client itself is implemented in Bourne shell the actual service instantiation modules can include any tools and/or languages available on the target platform. Pre- and post-processing hooks are also implemented to handle any setup or departure requirements needed for a sequence of service management requests.

Within this middleware solution the LDAP directory is defined as the official API for obtaining any and all information relevant to the implementation and delivery of services. This requirement enables the simplistic UAMS syntax. The UAMS client library provides a number of convenience functions for interrogating the LDAP directory and returning data attributes needed to implement the service instantiation process. This requirement has two implications: first of all, LAMS and UAMS invocations must be synchronized to insure that service creation occurs after the meta-directory system has populated or updated the LDAP directory servers. The second implication is that this requirement imparts an additional degree of security since an intruder would need to compromise the directory servers in order to effectively coerce the system into creating invalid or modified accounts.

KerDAP Enabled Applications

The primary goal of this middleware project was to implement authenticated and differentially authorized services to a wide variety of systems. The following services are currently implemented using the KerDAP API and under management of the USMS:

1. Generic host logins.
2. USENET news reading.
3. TACACS terminal server access.
4. WEB (Squid) proxy services.
5. IMAP email.

6. IMSP remote configuration management.
7. FTP file services.
8. Shared Message Block (SMB) file services.
9. WEB forms and applications.
10. User and system administration and management tools.

In almost all cases only minimal alterations to the sources were needed to implement the triad of IAA.

LDAP directory services were also heavily leveraged as a component of the clustering and high-availability strategies implemented in the data center. The most notable example of this is the use of IMAP and IMSP redirection systems. LDAP support was added to the open-source *perdition*[4] software as an alternative "database" for determining which server implements the IMAP message store and IMSP accounts for a user with email services. The mail destination attribute, which serves as the source for IMAP and IMSP redirections, also provides the basis for mail routing through the LDAP-enabled sendmail hubs which handle incoming mail for all organizational units serviced by this solution.

Future Initiatives

KerDAP is considered to be in a developmental phase of implementation. The USMS has demonstrated its ability to decrease administrative costs and to more precisely control the delivery of services to the user community. Experience with this initial implementation has led to plans for a number of important modifications to the middleware architecture.

First and perhaps foremost are plans for the implementation of an authorization server. Just as USDB, LAMS and UAMS provide the three components of IAA for the management system, the authorization server will couple with LDAP and Kerberos to complete the triad of IAA for KerDAP-enabled applications. The KerDAP library will be extended with a set of functions which will allow each service request to query an authorization server to determine whether or not access to the service is authorized at a particular instant in time. In this scheme the LDAP server assists the authorization process by supplying a unique service token for each service which has been bound to a service entity. This token is passed to the authorization server which than authorizes access to the service based on a set of rules which are specified generically for the service and more specifically for a particular service entity.

It is anticipated that this authorization server will allow the centralized management of IP access controls and other administrative functionality. Most importantly it will provide a mechanism for establishing finite service lifetimes and implementing the notion of organizational role playing for service entities within the enterprise structure. It is anticipated that this authorization server will play an important role either as a replacement or delivery mechanism for X.509v3 attribute certificates.

A second important area of anticipated development is in the merging of shared message block file (SMB) services with middleware services. Integration of the Samba file server with the middleware solution is anticipated to make this file sharing protocol more competitive with the management and administrative advantages of currently popular commercial alternatives. Initial work beyond simple authentication and authorization is focusing on allowing share configuration and access information to be obtained from the LDAP directory rather than host specific configuration files. A second and longer term project is to implement the notion of a Samba fanout server which would automatically redirect SMB connection requests based on user identification and the name of the requested file share.

Another important area of development is the delivery of KerDAP IAA services through the Pluggable Authentication Module (PAM) system. The PAM system is currently seeing widespread use throughout the Open-Source community. The goal of these efforts is to minimize the amount of modification needed at the source level of the service delivery applications.

Current trends of inter-operability in the network directory arena are being closely monitored as well. There are a number of efforts focusing on the use of XML and its derivatives to support the propagation of information from meta-directory systems into server directory databases. Of note is industry led work on the Directory Services Markup Language (DSML). If standards efforts

[4] http://perdition.sourceforge.net/

prevail the goal would be to implement the functionality of LAMS via DSML which would provide this middleware solution with a mechanism for propagating and replicating directory information into commercial as well as open-source directory server solutions.

Conclusions

The incredible success of the INTERNET, the increasing demand for mobile computing, and resultant business model paradigm shifts will demand middleware based IT architectures. The choice of a middleware solution will affect all other applications in the enterprise. Application and server systems will need to participate in and integrate with the middleware. Failure to do so will preclude their use within the framework of enterprise computing.

The Open Source and Linux communities must recognize this and respond with an open, standards-based solution to insure the continued viability of their service delivery platform. The success of the open source model to date indicates that it works and that, if an open middleware solution is developed, the INTERNET community will embrace it.

The success of KerDAP and USMS suggests that the notion of a viable middleware strategy based on an open system architecture and open source software is realistic. The authors are pleased with the interest in their solution and invite interested parties to contact them via email at ker_DAP@ndsu.nodak.edu. Copies of this paper are available via the web at http://www.ndsu.nodak.edu/servergroup/kerDAP/.

Acknowledgements

Any significant achievement is invariably a success due to individual contributions through a team effort. USMS and KerDAP is no exception to this rule.

The authors would like to first of all thank Ms. Bonnie Neas, Director of ITS at NDSU, for creating an environment which nourishes and encourages this type of effort in a research/production environment.

Special acknowledgements go to Mr. Dick Jacobson and Mr. Dale Summers. The development of the USDB and its integration with the administrative parent systems is the result of their programming and organizational talents. Success of any middleware initiative is dependent on access to individuals such as these that have a thorough knowledge of an organization, its operation and the policy implications of implementing effective middleware strategies. Their efforts were aided by Mr. Mick Pytlik and his administrative services staff which worked closely with them in identifying successful integration strategies.

Finally a special thanks to the tireless legions of Open-Source programmers. Their efforts toward the common, collective good provided the tools and the sandbox without which this project would not have been possible.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

Member Benefits:

- Free subscription to *;login:*, the Association's magazine, published eight times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and Open Source, book and software reviews, summaries of sessions at USENIX conferences, and Standards Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to *;login:* on the USENIX Web site.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, on the USENIX Web site.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as security, Linux, Internet technologies and systems, operating systems, and Windows—as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Savings on a variety of products, books, software, and periodicals: see *http://www.usenix.org/membership/specialdisc.html* for details.

Supporting Members of the USENIX Association:

Addison-Wesley
Earthlink Network
Edgix
Interhack Corporation
Interliant
JSB Software Technologies
Lucent Technologies
Macmillan Computer Publishing, USA
Microsoft Research
Motorola Australia Software Centre
Nimrod AS
O'Reilly & Associates Inc.
Performance Computing
Sendmail, Inc.
Smart Storage, Inc.
Sun Microsystems, Inc.
Sybase, Inc.
Syntax, Inc.
UUNET Technologies, Inc.
Web Publishing, Inc.

Supporting Members of SAGE:

Collective Technologies
Deer Run Associates
Electric Lightwave, Inc.
ESM Services, Inc.
GNAC, Inc.
Macmillan Computer Publishing, USA
Mentor Graphics Corp.
Microsoft Research
Motorola Australia Software Centre
New Riders Press
O'Reilly & Associates Inc.
Remedy Corporation
RIPE NCC
SysAdmin Magazine
Taos: The Sys Admin Company
Unix Guru Universe

For more information about membership, conferences, or publications,
see *http://www.usenix.org/*
or contact:
USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA.
Phone: 510-528-8649. Fax: 510-548-5738. Email: *office@usenix.org*.

ISBN 1-880446-17-0